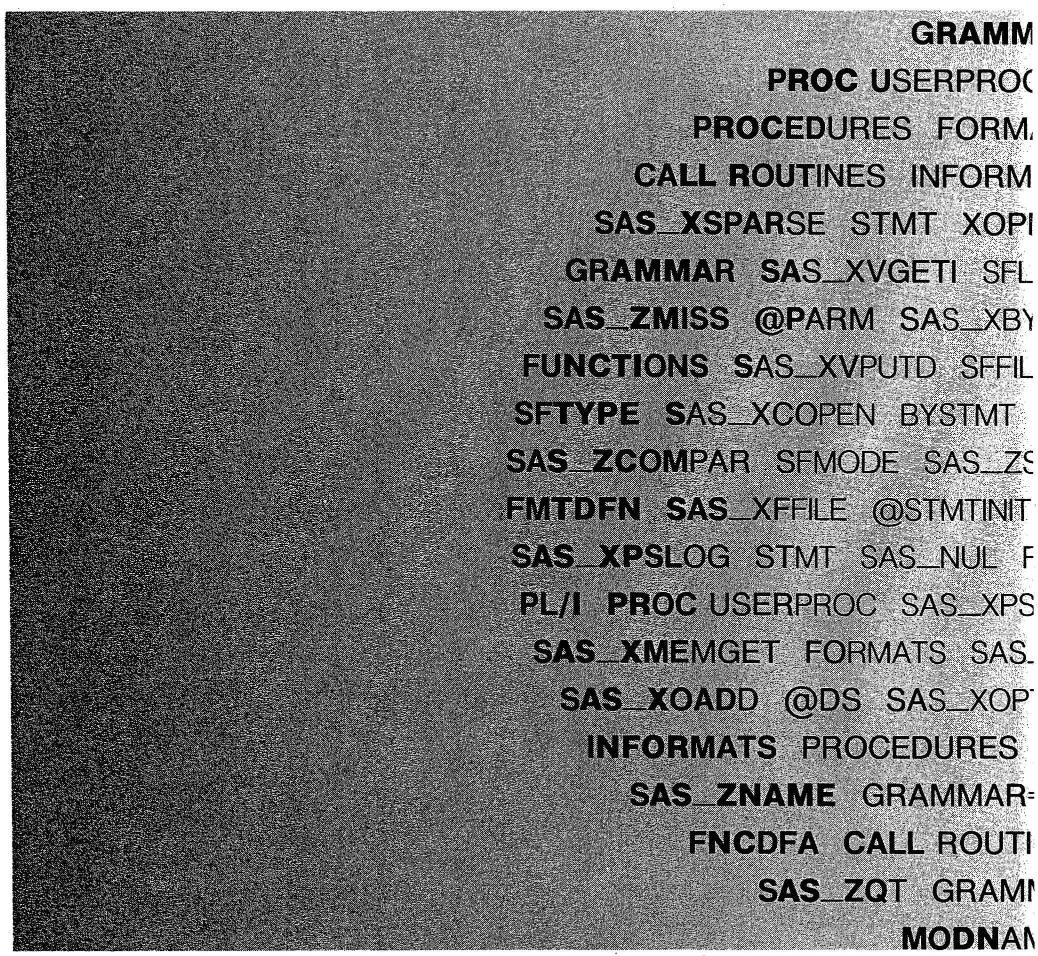
SAS/TOOLKIT™ Software

Usage and Reference
Version 6
First Edition



56049

SAS Institute Inc.

Overview of the SAS® System

Base SAS® Software
SAS® Language
Base SAS® Procedures
SAS® Macro Language
SAS® Display Manager System
SAS® Text Editor

Graphics Display
SAS/GRAPH® Software
SAS/REPLAY-CICS® Software

Applications Development
SAS/AF® Software
SAS/DMI® Software
SAS/TOOLKIT™ Software
SAS/FSP® Software
**Screen Control Language

Data Base and File Access Methods
SAS/ACCESS® Software
SAS/IMS-DL/I® Software
*SAS® Interface to VSAM
SAS/SHARE® Software

Statistical Design and Analysis
Econometrics and Forecasting
SAS/ETS® Software
SAS/INSIGHT™ Software
SAS/QC® Software
SAS/STAT® Software

Project Management and Operations Research
SAS/OR® Software

Matrix Language
SAS/IML® Software

Computer Performance Evaluation
SAS/CPE® Software
MXG® Software

Communication
SAS/RTERM® Software
SAS/CONNECT™ Software

User Interface Software
SAS/ASSIST® Software

*Included in the base SAS® software license.
**Included in SAS/AF® and SAS/FSP® software.

SAS/TOOLKIT™ Software: Usage and Reference

Version 6
First Edition

SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS/TOOLKIT™ Software: Usage and Reference, Version 6, First Edition,* Cary, NC: SAS Institute Inc., 1991. 761 pp.

SAS/TOOLKIT™ Software: Usage and Reference, Version 6, First Edition

ISBN 1-55544-453-9

1st printing, August 1991

Doc S19, Ver1.01, 072291

Contents

Part 3 ▪ Procedures 43

REFERENCE

Part 6 ▪ Specifications 267

Reference Aids

Figures

Tables

Credits

Documentation

Composition	Candy R. Farrell, Nancy Mitchell, Denise T. Jones, David S. Tyree
Graphic Design	Creative Services Department
Proofreading	Patsy P. Blessis, Carey H. Cox, Heather B. Dees, Josephine P. Pope, Toni P. Sherrill, John M. West
Technical Review	Stephen Beatrous, Tom Caulkins, Ronald Coggins, Tom Cole, Anne Corrigan, Ginny Dineley, Tricia Dudley, Georges H. Guirguis, Darylene C. Hecht, Amerie Helton, Brian Hess, F. W. Hester, Bonnie Horne, Paul J. Karlok, Richard D. Langston, Jean Moorefield, Mark Moorman, Elizabeth K. Norris, Ron Parker, Terry D. Poole, Joy Reel, Lisa M. Ripperton, J. Timothy Rowles, Douglas J. Sedlak, David Shamlin, W. David Shinn, Joseph G. Slater, Cheryl L. Turney, Maggie Underberg, Jade Walker, Randal K. Whitehead, Ken D. Worsham, Thomas W. Zack, Dea B. Zullo
Writing and Editing	Paramita Ghosh, Hanna P. Hicks, Kathryn A. Restivo, Helen Weeks, Holly S. Whittle

Software

The primary developer for SAS/TOOLKIT software is Richard D. Langston. In addition, Jack J. Rouse is responsible for the current implementation of the grammar processor. Jane Pierce contributed the SUBLIB procedure, which is provided as a sample procedure.

Using This Book

Purpose

SAS/TOOLKIT Software: Usage and Reference, Version 6, First Edition provides both usage and reference information about using SAS/TOOLKIT software with Release 6.07 and later releases of the SAS System.

The two types of information are presented in separate divisions of this book. The usage (or tutorial) division shows you how to perform the necessary tasks to create a procedure or a SAS informat, format, function, or CALL routine (IFFC). The reference division provides complete descriptions of all the routines provided with SAS/TOOLKIT software, describes special programming techniques for different languages, and provides information specific to each supported operating system.

This book is intended to provide complete information on SAS/TOOLKIT software. This book does not attempt to explain the SAS System or how standard SAS procedures work. For usage information on the SAS System, refer to *SAS Language and Procedures: Usage, Version 6, First Edition.* For reference information, refer to *SAS Language: Reference, Version 6, First Edition* and the *SAS Procedures Guide, Version 6, Third Edition.*

"Using This Book" describes how you can best use this book. It describes the book's intended audience, the audience's prerequisite knowledge, the book's organization and its conventions, and the additional SAS System documentation that is available to you.

Audience

This book is intended for programmers who are experienced in programming in the C, PL/I, FORTRAN, or IBM 370 assembler languages.

Prerequisites

The following table summarizes the concepts that you need to understand in order to use SAS/TOOLKIT software:

When you are	You need to understand	Refer to
any SAS/TOOLKIT user	how to invoke the SAS System at your site.	instructions provided by the SAS Software Consultant at your site
	concepts of SAS data libraries, SAS data sets, observations, and variables.	*SAS Language: Reference, Version 6, First Edition*
	concepts of SAS engines.	*SAS Language: Reference, Version 6, First Edition*
	the concept of a BY group.	*SAS Language: Reference, Version 6, First Edition*
	the SAS log and procedure output file.	*SAS Language: Reference, Version 6, First Edition*
	SAS system options and how they can affect procedure output.	*SAS Language: Reference, Version 6, First Edition*
	standard SAS formats, informats, functions, or CALL routines.	*SAS Language: Reference, Version 6, First Edition*
	how standard SAS procedures function.	*SAS Procedures Guide, Version 6, Third Edition*
	how to use the SAS System with your operating system.	the SAS companion for your operating system
writing your program in C	complex pointer resolution and multiple levels on indirection.	standard C programming reference books
	complex structures.	standard C programming reference books
	unions.	standard C programming reference books
writing your program in PL/I	BASED storage.	standard PL/I programming reference books
	pointers.	standard PL/I programming reference books

(*continued*)

(continued)

When you are	You need to understand	Refer to
writing your program in IBM 370 assembler	standard call-by-value calling sequences.	standard IBM 370 assembler programming reference books
writing your program in FORTRAN	that there are no special programming techniques required for FORTRAN. You must be an experienced FORTRAN programmer if you plan to use this language to write a SAS module.	

Before beginning this book, if you would like more details on the SAS System or the operating system you are using, refer to the documents listed in "Additional Documentation" later in this chapter.

Using SAS/TOOLKIT software also involves having

- SAS/TOOLKIT software, Release 6.07 or later.
- base SAS software, Release 6.07 or later.
- IBM's MVS or CMS operating systems or Digital Equipment Corporation's VMS operating system.
- the appropriate language compiler for the language you are using. Supported compilers are
  - the SAS/C Compiler under MVS and CMS
  - the native C compiler under VMS
  - the VS FORTRAN Version 3 compiler under MVS and CMS
  - the native FORTRAN compiler under VMS
  - the PL/I Optimizing Compiler under MVS and CMS
  - the native PL/I compiler under VMS
  - the IBM 370 Version 2 H-level Assembler.

How to Use This Book

This section gives an overview of the book's organization and content. In the first section, the book's divisions, parts, and chapters are described, and the following section, "What You Should Read," explains how to use each chapter.

Organization

SAS/TOOLKIT Software: Usage and Reference consists of two divisions that include eight parts.

Usage

The usage division of the book describes how to use SAS/TOOLKIT software to write procedures, informats, formats, functions, and CALL routines in any of the supported languages. This division also include instructions for converting older user-written procedures to Version 6 procedures.

Part 1: Introduction

Part 1 provides an overview of writing and using user-written procedures, informats, formats, functions, and CALL routines.

Chapter 1, "Introduction to SAS/TOOLKIT Software"

Part 2: Grammars

Part 2 introduces the concepts of the procedure grammars, discusses how to write a grammar, and provides instructions for using the grammar with the procedure.

Chapter 2, "Understanding Grammars"

Chapter 3, "Writing a Grammar"

Chapter 4, "Creating the Grammar Function"

Part 3: Procedures

Part 3 introduces the concepts of user-written procedures and illustrates how to write a procedure. This part includes a template for developing a procedure in each of the supported languages and an example in each language.

Chapter 5, "User-Written Procedures and the SAS System"

Chapter 6, "Building a Template for SAS/TOOLKIT Procedures in the C Language"

Chapter 7, "Building a Template for SAS/TOOLKIT Procedures in the FORTRAN Language"

Chapter 8, "Building a Template for SAS/TOOLKIT Procedures in the PL/I Language"

Chapter 9, "Building a Template for SAS/TOOLKIT Procedures in the IBM 370 Assembler Language"

Chapter 10, "Writing a SAS Procedure"

Chapter 11, "Sample Procedures in the FORTRAN, PL/I, and IBM 370 Assembler Languages"

Part 4: Informats, Formats, Functions, and CALL Routines

Part 4 introduces the concepts of user-written informats, formats, functions, and CALL routines and illustrates how to write each of these types of modules in all of the supported languages.

Chapter 12, "Understanding SAS Informats, Formats, Functions, and CALL Routines"

Chapter 13, "Writing a SAS Informat"

Chapter 14, "Writing a SAS Format"

Chapter 15, “Writing a SAS Function or CALL Routine”

Chapter 16, “Sample IFFCs in the FORTRAN, PL/I, and IBM 370 Assembler Languages”

Part 5: Converting Old Programs

Part 5 describes how to convert a procedure from Version 5 or Release 82.4 to Version 6 of the SAS System.

Chapter 17, “Converting from Version 5 to Version 6”

Chapter 18, “Converting from Release 82.4 to Version 6”

Reference

The reference division of the book provides complete reference information on SAS/TOOLKIT software.

Part 6: Specifications

Part 6 provides reference information for the USERPROC procedure and for the entire library of routines used to write procedures, informats, formats, functions, and CALL routines in any of the supported languages. This part also includes reference information on grammar coding and the data structures used with SAS/TOOLKIT software.

Chapter 19, “The USERPROC Procedure”

Chapter 20, “Data Structures”

Chapter 21, “Grammar Language”

Chapter 22, “SAS_X Routines”

Chapter 23, “SAS_Z Routines”

Chapter 24, “Miscellaneous Routines and Values”

Part 7: Language Dependencies

Part 7 contains helpful tips on writing efficient and portable programs in C, FORTRAN, and PL/I.

Chapter 25, “Writing Programs in the C Language”

Chapter 26, “Writing Programs in the FORTRAN Language”

Chapter 27, “Writing Programs in the PL/I Lanugage”

Part 8: Operating System Appendices

Part 8 describes how to use SAS/TOOLKIT software with each of the supported operating systems. These chapters contain instructions on compiling, linking, and running user-written SAS modules.

Appendix 1, “Using SAS/TOOLKIT Software under CMS”

Appendix 2, “Using SAS/TOOLKIT Software under MVS”

Appendix 3, “Using SAS/TOOLKIT Software under VMS”

What You Should Read

This book describes writing several different types of SAS modules in several different languages. The following table describes users and their needs, and indicates what parts of the book to read.

If you are	You should read
a programmer writing a SAS procedure in C	Chapter 1 to get an overview of how SAS/TOOLKIT software functions. Then read Chapters 2, 3, and 4 to develop the grammar for your procedure. Next, read Chapter 5 to learn about the concepts of writing SAS procedures. Follow the instructions in Chapter 6 for building a template for a C procedure; then skip to Chapter 10 for detailed instructions on performing the most common tasks in your procedure. Review the programming tips in Chapter 25 to ensure that your procedure is written efficiently and portably. Use the reference information in Chapters 19-24 as needed. Refer to the appropriate appendix for your operating system for instructions on compiling, linking, and running your procedure.
a programmer writing a SAS procedure in FORTRAN	Chapter 1 to get an overview of how SAS/TOOLKIT software functions. Then read Chapters 2, 3, and 4 to develop the grammar for your procedure. Next, read Chapter 5 to learn about the concepts of writing SAS procedures. Follow the instructions in Chapter 7 for building a template for a FORTRAN procedure; then skip to Chapter 10 for detailed instructions on performing the most common tasks in your procedure. Refer to Chapter 11 for a FORTRAN example of the procedure illustrated in Chapter 10. Review the programming tips in Chapter 26 to ensure that your procedure is written efficiently and portably. Use the reference information in Chapters 19-24 as needed. Refer to the appropriate appendix for your operating system for instructions on compiling, linking, and running your procedure.
a programmer writing a SAS procedure in PL/I	Chapter 1 to get an overview of how SAS/TOOLKIT software functions. Then read Chapters 2, 3, and 4 to develop the grammar for your procedure. Next, read Chapter 5 to learn about the concepts of writing SAS procedures. Follow the instructions in Chapter 8 for building a template for a PL/I procedure; then skip to Chapter 10 for detailed instructions on performing the most common tasks in your procedure. Refer to Chapter 11 for a PL/I example of the procedure illustrated in Chapter 10. Review the programming tips in Chapter 27 to ensure that your procedure is written efficiently and portably. Use the reference information in Chapters 19-24 as needed. Refer to the appropriate appendix for your operating system for instructions on compiling, linking, and running your procedure.

(*continued*)

(continued)

If you are	**You should read**
a programmer writing a SAS procedure in IBM 370 assembler language	Chapter 1 to get an overview of how SAS/TOOLKIT software functions. Then read Chapters 2, 3, and 4 to develop the grammar for your procedure. Next, read Chapter 5 to learn about the concepts of writing SAS procedures. Follow the instructions in Chapter 9 for building a template for an assembler procedure; then skip to Chapter 10 for detailed instructions on performing the most common tasks in your procedure. Refer to Chapter 11 for an IBM 370 assembler example of the procedure illustrated in Chapter 10. Use the reference information in Chapters 19-24 as needed. Refer to the appropriate appendix for your operating system for instructions on compiling, linking, and running your procedure.
a programmer writing a SAS informat, format, function, or CALL routine in C	Chapter 1 to get an overview of how SAS/TOOLKIT software functions. Then read Chapter 12 to learn about the concepts of writing SAS IFFCs. Follow the instructions in Chapter 13, 14, or 15, respectively, for writing informats, formats, or functions and CALL routines. Use the reference information in Chapters 19 and 24 as needed. Refer to the appropriate appendix for your operating system for instructions on compiling, linking, and running your IFFC.
a programmer writing a SAS informat, format, function, or CALL routine in FORTRAN, PL/I, or IBM 370 assembler	Chapter 1 to get an overview of how SAS/TOOLKIT software functions. Then read Chapter 12 to learn about the concepts of writing SAS IFFCs. Follow the instructions in Chapter 13, 14, or 15, respectively, for writing informats, formats, or functions and CALL routines. Refer to Chapter 16 for FORTRAN, PL/I, and IBM 370 assembler examples of the modules illustrated in Chapters 13, 14, and 15. Use the reference information in Chapters 19 and 24 as needed. Refer to the appropriate appendix for your operating system for instructions on compiling, linking, and running your IFFC.
converting a Version 5 procedure to Version 6	Chapter 1 and Chapter 5 to get an overview of how SAS/TOOLKIT software functions in Version 6. Read Chapter 17 for information on the changes that have occurred. Then refer to Chapter 10 and Chapters 20-24 as needed to make the changes to your procedure. Use Chapter 19 and the appropriate appendix for your operating system for instructions on running the USERPROC procedure and compiling, linking, and running your procedure.
converting a Release 82.4 procedure to Version 6	Chapter 18 for information on the changes that have occurred. This chapter helps you to see the relationship between tasks you performed in Release 82.4 and their corresponding tasks in Version 6. However, the concepts of procedure writing have changed so greatly from Release 82.4 to Version 6 that you need to follow the instructions for new procedure writers listed first in this table.

Reference Aids

The following reference aids provide more information about the concepts and terms used in this book. They are located at the end of the book.

Glossary	defines terms and concepts that are used in this book for SAS/TOOLKIT software and the SAS System.
Index	provides a cross-reference of the pages where specific topics and terms are discussed in this book.

The graphic on the inside front cover of this book provides a functional overview of the SAS System. The graphic on the inside back cover provides an overview of how SAS/TOOLKIT software relates to the SAS System and the specific needs at your site.

You can use these graphics as aids in determining which SAS software products you should use to accomplish particular data-management tasks.

Conventions

This section covers the conventions this book uses, including typographical conventions, usage conventions, and conventions used in presenting examples.

Typographical Conventions

In this book, you will see several type styles used. Style conventions are summarized here:

roman	is the basic type style used for most text.
UPPERCASE ROMAN	is used for references in the text to keywords of the SAS language, filenames, variable names, MVS JCL, CMS EXEC language, PL/I, FORTRAN, and IBM 370 assembler. Variable names from C language examples appear in uppercase in text only when they appear that way in the examples.
italic	is used for terms that are defined in the glossary and to emphasize important information.
`monospace`	is used to show examples of C or SAS programming code. In most cases, this book uses lowercase type for C programming statements and SAS code. Structure references and any variable names defined with the **`#define`** command are usually in uppercase monospace. Monospace is also used for C variable names that appear in text.
boldface	is used for the names of matrices and vectors. Matrices appear in uppercase bold; vectors appear in lowercase bold.

Conventions for Examples and Usage Sections

Most of the examples in this book are shown in the C language because that is the language in which most of the SAS System and SAS/TOOLKIT software are written. All of the SAS_X, SAS_Z, and miscellaneous functions included with SAS/TOOLKIT software use the same calling conventions for C and PL/I; for FORTRAN, the only difference is that statements do not end with a semicolon. For the SAS_X, SAS_Z, and miscellaneous routines that are CALL routines (not functions), the usage sections illustrate the correct calling sequence for each of these languages.

The only examples of IBM 370 assembler provided in this book are in the chapters that contain complete programs. The calling sequence for IBM 370 assembler should be familiar to any assembler programmer using this book and is too lengthy to show in detail in each of the usage sections.

When an example is labeled as a **Listing**, the example is included on the installation tape for SAS/TOOLKIT software.

Using the SAS System

This book does not attempt to describe how to use the SAS System in detail. Each of the operating system appendices provide sample statements for running the SAS System to test your SAS module. Note that once you have created a procedure or other SAS module, you can run your SAS module using any method of running the SAS System, including the SAS Display Manager System. For more information on running the SAS System, refer to the SAS companion for your operating system.

Additional Documentation

You may find the following documentation helpful when you are using SAS/TOOLKIT software and the SAS System.

SAS Documentation

There are many SAS System publications available. To receive a free *Publications Catalog*, write to the following address:

SAS Institute Inc.
Book Sales Department
SAS Campus Drive
Cary, NC 27513

The books listed here should help you find answers to questions you may have about the SAS System in general or specific aspects of the SAS System.

- *SAS Language and Procedures: Introduction, Version 6, First Edition* (order #A56074) gets you started if you are unfamiliar with the SAS System or any programming language.
- *SAS Language and Procedures: Usage, Version 6, First Edition* (order #A56075) is a user's guide to the SAS System. It shows you how to use base SAS software for data analysis, report writing, and data manipulation. It also includes information on methods of running the SAS System and accessing SAS files.
- *SAS Language: Reference, Version 6, First Edition* (order #A56076) provides detailed information on base SAS software, the SAS programming language, and the types of applications the SAS System can perform. Chapter 6, "SAS Files," explains how the implementation of SAS data sets has changed in Version 6 of the SAS System.
- *SAS Procedures Guide, Version 6, Third Edition* (order #A56080) provides detailed information about the procedures available within base SAS software.
- *SAS Companion for the CMS Environment, Version 6, First Edition* (order #A56103) provides detailed information on running the SAS System under CMS.
- *SAS Companion for the MVS Environment, Version 6, First Edition* (order #A56101) provides detailed information on running the SAS System under MVS.
- *SAS Companion for the VMS Environment, Version 6, First Edition* (order #A56102) provides detailed information on running the SAS System under VMS.

USAGE

Part 1
Introduction

Chapter 1 Introduction to SAS/TOOLKIT™ Software

Introduction

SAS/TOOLKIT software enables you to write customized SAS procedures, informats, formats, functions, and CALL routines in any of these languages: C, FORTRAN, PL/I, or IBM 370 assembler. This chapter describes the general capabilites of SAS/TOOLKIT software and provides a quick overview of how to use the product.

Overview of Writing SAS Procedures

A user-written SAS procedure is a program written in C, FORTRAN, PL/I, or IBM 370 assembler that interfaces with the SAS System to perform a given action. The SAS System provides services to the procedure, such as statement processing, data set management, and memory allocation. SAS procedures read SAS data sets, perform data analysis, print results, and create other SAS data sets.

The following sections explain why you may want to write a customized SAS procedure and provide a brief overview of the necessary steps.

Why Write a SAS Procedure?

There are many advantages to writing programs as SAS procedures rather than as stand-alone programs:

- Using your new procedure is easy for SAS users because they already understand data input, data manipulation, and general SAS syntax.
- You can extend the capabilities of the SAS System to meet the specialized need of your users.
- A program incorporated into the SAS System enables you to put all your programming tools in one place.
- The SAS System provides many services that you, as the programmer, would otherwise have to code yourself.
- The data used by your procedure can easily be sorted, printed, and analyzed using other SAS procedures during a single job.
- Special features, like BY-group processing and WHERE processing, can be used.
- The task of documenting your new program is simplified because only the new language statements, the procedure's output, and any special calculations need to be explained.

Steps in Writing a SAS Procedure

If you have already decided to write a SAS procedure, you probably have in mind what you want your procedure to do. Determining the function and purpose of the procedure and planning your program are important preliminaries to writing SAS procedures.

Writing the Grammar

One of the important parts of planning your procedure is deciding how you want the user to invoke your procedure. You must have this information to write the *grammar* for the procedure. The grammar defines the format of the PROC statement, any other statements that can be used when invoking your procedure, and any options that can be used in the PROC or other statements.

The SAS System uses the syntax rules you define in your grammar to check the syntax of the statements when the user invokes your procedure.

Writing the Procedure

The biggest task of writing the procedure is coding the application program that actually performs the function of the procedure.

To assist you in coding your procedure, SAS/TOOLKIT software provides a library of routines that enable you to interface with the SAS System and perform other frequently needed tasks.

The library routines are grouped into sets of routines that perform related actions. For example, there are sets of routines for all of the following tasks:

- performing I/O on SAS data sets
- dynamically allocating memory for use by your program
- printing procedure output or messages to the SAS log
- providing detailed information on observations and variables in SAS data sets
- handling problems specific to the SAS environment, such as missing values
- replacing routines standard to the programming language you are using when the standard routines do not meet the needs of the SAS System.

Using the USERPROC Procedure

Before you can run your grammar or your procedure, you must use the USERPROC procedure of SAS/TOOLKIT software. For grammars, the USERPROC procedure produces a grammar function. For procedures, the USERPROC procedure produces a program constants object file, which is necessary when linking all of your compiled object files into an executable module.

Compiling and Linking the Procedure with the SAS System

You must compile and link the program constants object module, the output of PROC USERPROC, with your procedure and store the executable program in a library that the SAS System can access when a user invokes your procedure. Compiling and linking are tasks that are specific to each operating system; refer to the appendix specific to your operating system for instructions on these processes.

Overview of Writing SAS IFFCs

In addition to writing SAS procedures, you can use SAS/TOOLKIT software to write your own SAS informats, formats, functions, and CALL routines (IFFCs) in the same choice of languages: C, FORTRAN, PL/I, or IBM 370 assembler. The following sections explain why you may want to use SAS/TOOLKIT software for creating IFFCs and briefly explain the necessary steps.

Why Write a SAS IFFC?

Like procedures, user-written functions and CALL routines add additional capabilities to the SAS System that enable you to tailor the system to your site's specific needs. Many of the same reasons for writing procedures also apply to writing SAS functions and CALL routines.

Formats and informats also enable you to tailor the SAS System to your needs, but you may wonder why you should use SAS/TOOLKIT software to create user-written formats and informats when base SAS software includes the FORMAT procedure. Using SAS/TOOLKIT software enables you to create formats and informats that perform more than the simple table lookup functions provided by the FORMAT procedure. When you write formats and informats with SAS/TOOLKIT software, you can do the following:

- assign values according to an algorithm instead of looking up a value in a table. You can include calculations and other algorithmic capabilities in your formats and informats that you write with SAS/TOOLKIT software.
- look up values in a database to assign formatted values. You cannot do this with formats created with the FORMAT procedure.

Steps in Writing a SAS IFFC

Unlike SAS procedures, IFFCs do not require that you define a grammar. The steps to writing a SAS IFFC are covered in the following sections.

Writing the IFFC

In general, IFFCs are simpler and shorter than user-written procedures. You do not need to interface with the SAS System as much in an IFFC, but when you do, you can use the same library of routines that are available with procedures. The routines you are most likely to use when writing an IFFC perform the following tasks:

- printing procedure output or messages to the SAS log
- handling problems specific to the SAS environment, such as missing values
- replacing routines standard to the programming language you are using when the standard routines do not meet the needs of the SAS System.

Using the USERPROC Procedure

Before you can compile your IFFC, you must use the USERPROC procedure of SAS/TOOLKIT software. For IFFCs, the USERPROC procedure produces a program constants object file, which is necessary when linking all the object files into an executable module.

Compiling and Linking the IFFC with the SAS System

You must compile your source, link the objects, and store the resulting executable program in a library that the SAS System can access when a user invokes your IFFC. Compiling and linking are tasks that are specific to each operating system; refer to the appendix specific to your operating system for instructions on these processes.

Using User-Written Procedures and IFFCs

After you have created a SAS procedure or IFFC, you need to tell the SAS System where to find the module when you want to run it. You can store your executable modules in any appropriate library or directory. Before you invoke the SAS System, use your operating system control language to identify where your executable modules are stored. (This process is described in detail in the appropriate appendix for your operating system.) When you invoke the SAS System and use the name of your procedure or IFFC, the SAS System checks its own storage locations first and then looks in the library or directory you identified for user-written modules for a module with that name.

Consult with the SAS Software Representative at your site to learn if there is a standard location for storing user-written modules at your site.

Part 2
Grammars

Chapter 2 Understanding Grammars

Introduction

A necessary part of writing a SAS procedure is defining the statements that can be used to invoke the procedure. Certain design decisions must be made concerning the format of these statements:

- What arguments will be available to the user and what mnemonic name (or names) will they have?
- Which arguments will be bit (on/off) arguments and which will be parameters, arguments that have values assigned to them? For the parameters, what types of values are acceptable?
- Will your procedure require any input SAS data sets or create any output data sets?
- Which standard SAS procedure statements (for example, VAR, ID, or WEIGHT) will be permitted, and what types of variables (numeric, character, or both) can be specified in these statements?
- Do you need to create special procedure statements other than the standard SAS procedure statements?

This chapter describes how to implement the format of the statements that you plan to use with your procedure.

Communicating with a SAS Procedure

Many standard programs, that is programs that are not SAS procedures, are written to handle a specific file. When you write these types of programs, you know exactly what variables are in the file, and you write your program so it works specifically with the contents of that file. This is how most DATA step programs work. To code the INPUT statement, the programmer must know the layout of the external file.

Most SAS procedures, however, are written so that they can process any SAS data set with virtually any number or type of variables. Before a SAS procedure can begin to manipulate the data in a SAS data set, it must determine which data set and variables the user wants to process. So, the first task your SAS procedure has is finding out what it is supposed to do!

The SAS System uses the PROC statement and subordinate procedure statements to enable the user to specify how a procedure should work. That is, most SAS procedures include a DATA= option in the PROC statement to enable the user to specify which SAS data set to process. Then, many SAS procedures permit a VAR statement so that the user can specify which variables in the input data set should be processed. Neither of these tools, the DATA= option or the VAR statement, are required, but they are useful methods of communicating with the SAS procedure. If you want your procedure to employ these tools, however, you must define them to the SAS Supervisor, the internal managing program of the SAS System. To describe these tools and other statements and options that can be used to communicate information into your procedure, you define a grammar.

What Is a Grammar?

A *grammar* is a set of rules that define the meaningful forms of a language. The grammars of spoken languages define parts of speech and how to combine them to form sentences. In terms of writing procedures for the SAS System, a grammar is the set of rules that describe how a user can invoke the procedure. The grammar you define serves two purposes:

- It defines what statements and arguments, such as options, parameters, and data set references, are permitted by your procedure.
- It also describes how to interpret and store the information in the statements specified by the user.

For example, you could use the following statements to invoke the PRINT procedure:

```
proc print data=a.sample  double;
   var item1 item2;
```

These statements are analyzed according to the instructions in the grammar for PROC PRINT. First the system checks the syntax. It ensures that both the DATA= and DOUBLE options are permitted in the PROC PRINT statement and that the VAR statement is a valid statement to use with the PRINT procedure.

The grammar is also used to interpret the information and perform appropriate actions. That is, the grammar indicates that A.SAMPLE is a reference to a SAS data set so the system opens the data set. The grammar also knows that the words following the VAR statement name are variable names. The system looks into the DATA= data set and verifies that all the variables named in the VAR statement exist in the data set.

From this PROC PRINT example, you can see that the grammar for a procedure permits you to establish the vocabulary for the procedure and perform some simple error checking.

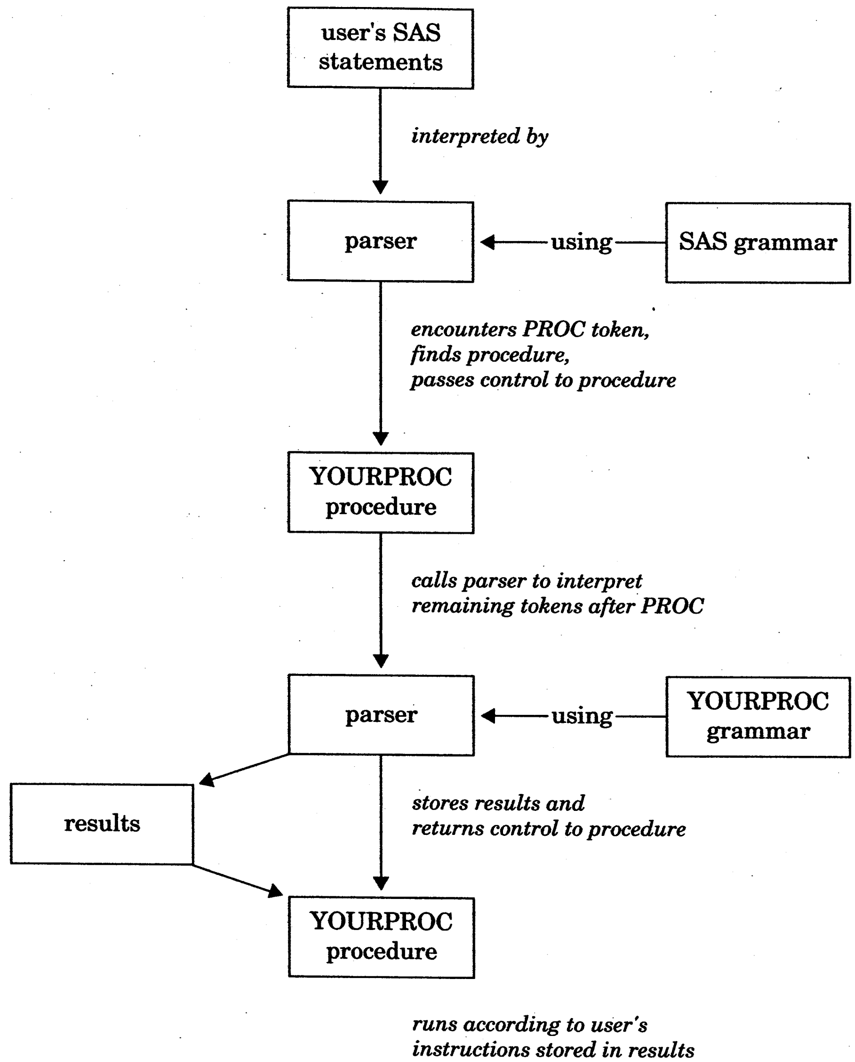
How the SAS System Uses Grammars

Each part of the SAS System has a grammar that defines how to interpret statements entered by the user. The grammars are used by the *parser*, a part of the SAS Supervisor that breaks each SAS statement into single tokens. A *token* can be a word, an operator, or a symbol. The parser examines each token and uses the grammars in the SAS System to decide what action to take for that token.

For example, the word PROC is a token in one of the primary grammars of the SAS System. When the parser encounters this word, it knows that the user is invoking a SAS procedure. The parser looks at the next token to determine which procedure to process. The parser validates the procedure name and looks for a SAS procedure with that name. If the name of the procedure is not one of the standard procedures included with the SAS System, the parser looks for a user-written procedure with that name.

When the parser finds the appropriate procedure, it passes control to the procedure. One of the first actions the procedure takes is to call the parser again. When the procedure calls the parser, it tells the parser to use the procedure's grammar to validate the remaining statements in the PROC step. The parser then validates the statements, interprets them, stores the information that is in the statements, and returns control to the procedure. Figure 2.1 illustrates this process.

Figure 2.1
Using Grammars to Process Statements



The parser uses your grammar to accomplish two major tasks for the procedure: validating the statements that invoke the procedure and creating data structures to store information for the procedure. Each of these functions is described in the following sections.

Validating Statements

The parser validates the statements used to invoke your procedure one at a time. The parser uses your grammar to

□ ensure that only the statements you defined and the statements that are permissible anywhere in the SAS System were used to invoke your procedure.

□ verify that arguments in statements are entered as described in your grammar.

- ensure that the variables listed in statements are valid variables and that they are the correct type required by the rules of your grammar. For example, if the rules of your grammar require that only numeric variables can be used in the WEIGHT statement, the parser accesses the input SAS data set, verifies that the listed variables exist in the data set, and looks up the type of each variable specified in the statement to ensure that it is numeric.
- check input values against a range of values defined in your grammar. For example, if you have defined an argument that can have only a few values, the parser can verify that the user specified one of the valid values.

Although the parser checks the syntax of each statement, it does not check for the order in which the statements appear. If you require that some statements be specified in a particular order, you must check the order in your procedure.

Storing Data Provided in SAS Statements

Many of the statements the user specifies in calling a procedure contain information that has to be stored and passed to your procedure. For example, if your procedure permits a VAR statement or a BY statement, the parser needs to pass the list of variables specified in these statements to your procedure. The parser stores information contained in SAS statements in a series of data structures called *statement structures*. Your procedure then accesses these structures to determine which arguments and statements were invoked and to access lists of variables and other data.

Chapter 5, "User-Written Procedures and the SAS System," explains in more detail how to access this stored information.

Steps for Creating and Using Grammars

The following sections summarize what you must do to create and use a grammar and describe the grammar code you write. Instructions for writing grammars are provided in Chapter 3, "Writing a Grammar."

Overview of the Process

When you write a SAS procedure, you must also create a grammar for the procedure. You write the grammar using a special grammar language. Steps for writing a grammar are explained in detail in Chapter 3. When you have finished writing the grammar, you convert the grammar to a function in the same language that you are using to write your procedure. That is, if you are writing a procedure in C, you convert your grammar code to a C function that can be called by your procedure. Chapter 4, "Creating the Grammar Function," details how to create the grammar function.

Once you have written your procedure, written your grammar, and converted it to a grammar function, you can compile and link these modules together to create the SAS procedure.

Grammar Coding

Most of the grammar code for a procedure is already defined for you in a module called the *stub grammar*. You include this module at the beginning of your grammar, and then you only need to define the statements and the arguments that are specific to your procedure. You define these portions of the grammar by using *productions*, which are rules that are applied during the parsing process. For example, a production can define the syntax for a statement used to invoke your procedure. A production consists of two parts, the name and the rule:

```
production name=rule,
```

Note that the rule ends with a comma. (The end of your entire grammar is signaled by a period.)

The name of the production can be any descriptive name of 32 characters or less. This name has no significance outside of your grammar, but be sure not to use one of the names already assigned to the standardized productions in the stub grammar. The rules in the production can be a combination of the following:

- *terminals*, which are the terms that the user specifies to invoke a statement or argument. When you define the grammar, you can choose any term or set of alternative terms, but when the procedure is invoked, the user must specify the exact term you defined. Terminals are always defined by enclosing the term in double quotation marks. When the user specifies a terminal, the actions that are associated with the terminal in the grammar are performed.
- *lexicals*, which are primitive word types such as numbers or names. Lexicals define restrictions on the type of input for the statement.
- *semantic actions*, which signal the parser to perform some action on the terminal or lexical associated with the semantic action. Semantic actions always begin with an at sign (@).
- *operators*, which indicate the relationship between the other components of the rule.
- other production names defined in either the stub grammar or your own grammar.

Note: Unfortunately, lexicals and production names do not have distinguishing punctuation marks to identify them (as terminals have quotation marks and semantic actions have @). If you are not certain which type of code appears in one of the samples in this chapter, refer to the lists of lexicals and production names in Chapter 21, "Grammar Language."

Chapter 3 Writing a Grammar

Introduction

This chapter builds on the concepts of grammar processing discussed in the last chapter by illustrating how to write a simple grammar. To better explain how to write a grammar, the chapter shows the syntax for a fictional procedure and then explains how to build the grammar for that procedure.

Before reading this chapter, you should be familiar with the following concepts and terms from the previous chapter:

- stub grammar
- productions
- semantic actions
- terminals
- lexicals
- operators.

Syntax for a Sample Procedure

This chapter uses the following sample procedure to show how to build a grammar:

PROC TESTIT <DATA=*SAS-data-set*> <OUT=*SAS-data-set*>
<HIVAL=*max* LOVAL=*min*> <YEARLY | MONTHLY>;
VAR *variable-list;*
BY *variable-list;*
COMPARE *variable=value-list;*

The PROC TESTIT statement has the following arguments:

DATA=*SAS-data-set*
: names the input data set. If the DATA= option is not specified, the TESTIT procedure uses the most recently created SAS data set.

HIVAL=*max*
: specifies a maximum numeric value for testing.

LOVAL=*min*
: specifies a minimum numeric value for testing.

OUT=*SAS-data-set*
: names the output data set. If the OUT= option is not specified, the TESTIT procedure creates a temporary SAS data set.

Note: Most SAS procedures that permit an OUT= option do not function like this one. In most cases, when the OUT= option is omitted, no data set is created. In this example a temporary data set is created.

YEARLY | MONTHLY
: are mutually exclusive options that indicate a time period should be used for testing. If neither of these options is specified, no limit is placed on the time period.

In addition to the PROC TESTIT statement, the procedure includes the following statements:

VAR
: is optional and specifies which variables should be tested by the procedure. The variables in the VAR statement should all be numeric. If the VAR statement is omitted, the grammar does not perform any processing for variables.

Note: The procedure associated with these statements can create a default list of variables, but the grammar does not automatically create this list.

BY
: is optional and lists BY variables.

COMPARE
: is also optional and has a variable name followed by a list of numeric values for comparison.

Defining the Syntax

The grammar for a procedure performs two major functions: it describes the syntax for the statements and arguments that can be used with the procedure, and it specifies how to store the information provided by the user when the procedure is invoked. This section explains how to build the grammar so it enforces the syntax of the procedure as it is described in the previous section. Keep in mind that defining the syntax is only part of building a grammar. You can add the other parts of the grammar, the semantic actions, after you define the syntax for the procedure.

In discussing the syntax of the grammar, you need to keep in mind the following terms:

production
: is a rule of the grammar. It consists of a production name and the rule that is associated with that production name.

terminal
: is the literal string that the user specifies to invoke a statement or argument. Terminals are always enclosed in double quotation marks.

operator
: is a symbol indicating the relationship between parts of a rule.

Minimum Requirements for a Grammar

In the current release of SAS/TOOLKIT software, all grammar statements must be coded in uppercase letters. However, this does not mean that the user must use uppercase letters to invoke your procedure, because SAS statements entered by the user are automatically uppercased when they are parsed (with the exception of quoted strings).

Note: Although comments are not required, you can make your grammar more readable with comment lines. A comment line must begin and end with a pound sign (#), as shown here:

```
#------------------------------ Sample Grammar -----------------------#
```

All grammars must include the following three statements:

```
%INCLUDE "STUBGRM" .
PROGRAM = ANYSTMT ENDJB ,
ANYSTMT = permitted-statements,
```

The first statement includes the stub grammar, the file that contains definitions for standard productions that you use in defining your procedure statements.

```
%INCLUDE "STUBGRM" .
```

This stub is provided with SAS/TOOLKIT software. Note that all %INCLUDE statements must end with a period.

The next statement is the production that initializes your grammar.

```
PROGRAM = ANYSTMT ENDJB ,
```

Note the comma at the end of this production. All productions except the last are ended by commas. The last production in the grammar is terminated with a period.

The next production declares the statements that your procedure can have. In most grammars, this statement takes the following form:

```
ANYSTMT = proc-statement
        | other-statement-1
        | ...
        | other-statement-n,
```

Note that the ANYSTMT production uses the OR operator (|) to separate the permitted statements. The elements connected by this operator are called *alternations*. An alternation consists of a group of terminals, lexicals, or production names; a single alternation can occur at a time. So in the ANYSTMT production, only one statement can be processed at a time. For the sample procedure, the ANYSTMT production is as follows:

```
ANYSTMT = TESTITSTMT
        | VARSTMT
        | BYSTMT
        | COMPARESTMT ,
```

The ANYSTMT production consists of four alternations: TESTITSTMT, VARSTMT, BYSTMT, and COMPARESTMT. As you may notice, the ANYSTMT production does not list the actual names of the statements in the procedure; instead, it lists the names of the productions in the grammar that handle each statement in the procedure.

How the SAS System Uses the Grammar

As explained in Chapter 2, "Understanding Grammars," when the SAS System reads the statement that invokes a procedure, it passes control of the parser to that procedure. The grammar for the procedure provides instructions for parsing statements until a new DATA step, PROC step, or the end of the job is encountered. If the user specifies a NULL statement, a global SAS statement (such as an ATTRIBUTE or WHERE statement), a macro definition, or a comment while using your procedure, those statements are automatically handled by the SAS System; the grammar for your procedure does not need to handle them.

The parser uses the two productions discussed so far to validate the statements specified with your procedure. Your grammar is processed for every statement used with the procedure. The ENDJB production in the PROGRAM production is defined in the stub grammar to be a semicolon. Thus, the end of every statement is a semicolon. The ANYSTMT production defines which statements are valid with your procedure. During any one execution of the grammar, the parser can encounter any of the statements listed in the ANYSTMT production.

For example, for this grammar when the parser encounters the PROC TESTIT statement, it processes that statement through the grammar. If the user also specifies the BY statement, the parser processes that statement in a separate pass through the grammar.

```
PROGRAM = ANYSTMT ENDJB ,

ANYSTMT = TESTITSTMT
        | VARSTMT
        | BYSTMT
        | COMPARESTMT ,
```

Expanding the Syntax for the Procedure

Each of the statements declared in the ANYSTMT production needs to be more fully described in the remainder of the grammar. This section explains how to describe statement names, bit arguments, and parameters that can be used with statements.

Note: In standard SAS documentation, all the syntax items that follow the statement name are called *arguments*. When you are writing your own procedure, it is useful to distinguish between two types of arguments: bit arguments and parameter arguments, or more simply parameters. *Bit arguments* are flags that are turned on by specifying the name of the argument. *Parameters* require a value in addition to the name of the argument.

Defining the Procedure Statement

Begin by defining the PROC TESTIT statement. Note that the keyword PROC is not included in the production for this statement because that keyword is handled separately by the parser. According to the description of the procedure at the beginning of the chapter, the PROC TESTIT statement permits several optional arguments. The following production describes the PROC TESTIT statement:

```
TESTITSTMT = "TESTIT"      TESTOPTS* ,
```

The TESTITSTMT production consists of the "TESTIT" terminal followed by another production, TESTOPTS. The TESTOPTS production appears later in the grammar and defines each of the arguments available in the TESTIT statement. The asterisk (*) is an operator indicating that the TESTOPTS production may occur zero or more times. That is, none or some or all of the arguments listed in TESTOPTS can be specified in the PROC TESTIT statement. This allows for any of the following invocations (among other possibilities):

```
PROC TESTIT;
PROC TESTIT DATA=X;
PROC TESTIT YEARLY;
PROC TESTIT DATA=X OUT=Y HIVAL=20 MONTHLY;
```

The TESTOPTS production is covered in more detail later in this chapter.

Defining the VAR Statement

The VAR statement in this procedure permits a list of numeric variables. The VARSTMT production defines the syntax for the VAR statement as follows:

```
VARSTMT     =  ("VARIABLES" | "VAR")    VARLIST ,
```

Note: You can embed any number of blank spaces within the grammar statements. Because the productions that you see here are incomplete (at this point they only describe the syntax, not the semantic actions), a number of the blank spaces in the production shown here will be filled in later.

In this example, the VARSTMT production indicates that the VAR statement can begin with the keyword VARIABLES or VAR, followed by whatever the VARLIST production expands into. Note that the VARIABLES and VAR keywords are alternation items, since they are separated by the OR operator (|). They are enclosed in parentheses to indicate that only those two terminals form the alternation items. Suppose the production did not include parentheses, as shown here.

```
#-----------------WRONG----------------------------#
VARSTMT     =  "VARIABLES" | "VAR"      VARLIST ,
```

The alternation items would be

```
"VARIABLES"
```

and

```
"VAR" VARLIST
```

In this case, the VARLIST production is associated only with one alias for the statement because the OR operator has lower precedence than the blank space, the operator that concatenates VAR and VARLIST. The best rule of thumb is, whenever in doubt, use parentheses.

Now consider the VARLIST production. VARLIST is a production defined in the stub grammar that was included earlier in the grammar. It allows any valid SAS variable list, such as any of the following:

```
A
A1-A10
A:
A--B
A-NUMERIC-B
_CHARACTER_
A B C X1-X10 Y:
```

The stub grammar includes a number of productions such as the VARLIST production that can help you define common SAS statements in your procedure. Before writing your procedure, study the standard productions described in Chapter 21, "Grammar Language."

As you remember, the variable list for this procedure permits only numeric variables. This requirement for the VAR statement is actually defined at the time the semantic actions are defined for the grammar, so at this point, the syntax does not include the restrictions for numeric variables. You'll see how to limit the variable list to numeric variables later in this chapter.

Defining the BY Statement

The BY statement is a standard SAS statement like the VAR statement; in fact, it is more standardized than the VAR statement. While a VAR statement can differ by permitting only numeric variables or by permitting only one variable, the BY statement doesn't have these differences. Because the BY statement is so standardized, the definition for the standard BY production, BYSTMT, is included in the stub grammar. When you include BYSTMT in the ANYSTMT production, you have done all that is necessary for you to be able to use the BY statement with your procedure.

Additional Statements

The last statement for the sample procedure, COMPARE, is not a standard type of SAS statement. Your grammar needs to define it in detail. The following production describes the syntax for the COMPARE statement:

```
COMPARESTMT =  "COMPARE"                 ONEVAR     "="
                                         NUMLIST ,
```

The "COMPARE" terminal, which defines the statement name, is followed by the ONEVAR production. This production is defined in the stub. It is similar to the VARLIST production but allows only one variable to be listed. The next symbol ("=") is another terminal, which indicates that the variable must be followed by an equal sign. The NUMLIST production indicates that the statement can have lists of numbers. As mentioned in the VAR statement, the limitations on the types of values permitted in this list are not included in the grammar until you add the semantic actions. At this point, the syntax simply permits lists of values.

Defining Statement Arguments

Many procedures permit bit arguments and parameters in statements. In the sample procedure, the PROC TESTIT statement permits the following arguments:

- DATA=*SAS-data-set*
- HIVAL=*max*
- LOVAL=*min*
- OUT=*SAS-data-set*
- YEARLY | MONTHLY

Remember that the production for the PROC TESTIT statement has already been defined as follows:

```
TESTITSTMT = "TESTIT"      TESTOPTS* ,
```

You can expand on the TESTOPTS production to define each of the arguments permitted in the PROC TESTIT statement. Note the period at the end of this production since it is the last production in the grammar.

```
TESTOPTS     = ("DATA"              "="    DSFIELD)
             | ("OUT"               "="    DSFIELD)
             |(("HIVAL" | "HI")     "="    INT)
             |(("LOVAL" | "LO")     "="    INT)
             |  "YEARLY"
             |  "MONTHLY"
             .
```

Both the DATA= and OUT= arguments are defined as a combination of two terminals, the name of the argument ("DATA" or "OUT") and the equal sign, ("="). The equal sign needs to be separate from DATA and OUT because the SAS parser breaks it out separately even if the equal sign follows immediately after the argument name without any blanks.

In addition, both the DATA= and OUT= arguments are followed by the DSFIELD production, which is defined in the stub grammar. The DSFIELD production allows for the proper parsing of the SAS data set specification as well as any SAS data set options. When you use this production, your grammar can process any valid SAS data set reference, as illustrated in the following examples:

```
x
x.y
x (firstobs=5)
x.y(keep=a rename=(a=b) obs=7)
_last_
_data_
_null_
```

The HIVAL and LOVAL parameters are defined by grouping the two possible terminals for the argument in parentheses followed by the terminal for the equal sign. The next item that appears in the definition of these parameters is the INT lexical. As mentioned in Chapter 2, lexicals define the type of a single value. In this case, both the HIVAL= argument and the LOVAL= argument permit a single integer value. The following values could be specified for either of these arguments:

- 1
- 10000000
- −5555
- 0

The following values are not valid for values defined with the INT lexical even though they represent integral values (100, 150, and 1), because they are expressed in decimal or scientific notation:

- 1E2
- 1.5E2
- 1.0

Note: If you want to specify numeric values of any type, use the NBR lexical. Refer to the description of the available lexicals in Chapter 21 for more information on defining types of data.

The Complete Syntax

The following example summarizes the sample grammar so far. In addition to the statements you have already seen, this sample includes comments. Comments start with a # symbol and end with a # symbol or the end of the line, whichever comes first. In the next section, you will add semantic actions to the following syntax:

```
#------------------------------ Sample Grammar -----------------------#
PROGRAM = ANYSTMT ENDJB ,

ANYSTMT = TESTITSTMT
        | VARSTMT
        | BYSTMT
        | COMPARESTMT ,


#------------------------- Statement Productions -------------------#

TESTITSTMT = "TESTIT"      TESTOPTS* ,


VARSTMT    = ("VARIABLES" | "VAR")   VARLIST ,


COMPARESTMT = "COMPARE"              ONEVAR     "="
                                     NUMLIST ,

#------------------------ PROC Statement Options -------------------#

TESTOPTS    = ("DATA"              "="   DSFIELD)
            | ("OUT"               "="   DSFIELD)
            |(("HIVAL" | "HI")     "="   INT)
            |(("LOVAL" | "LO")     "="   INT)
            |  "YEARLY"
            |  "MONTHLY"
            .

#--------------------------- End of Grammar ------------------------#
```

Adding Semantic Information

So far, the grammar coding shown in this chapter has defined the syntax of the statements used with the procedure. The grammar has another function; it also specifies how to store and, in some cases, verify the information provided by the user when the procedure is invoked. The instructions in the grammar for storing and verifying the information provided by the user are called *semantic information.*

The semantic information in the grammar passes this kind of information to the procedure:

- What options did the user specify in the PROC statement?
- Which statements did the user specify?
- If the BY and VAR statements are specified, what variables are listed in those statements?
- If the COMPARE statement is specified, what variable is named and what are the values in the value list?

In addition, the semantic information can perform verifications like the following:

- Are the variables listed in the VAR statement all numeric variables?
- Are the values listed in the COMPARE statement all numeric values?
- Are the names of data sets for the DATA= and OUT= options valid formats for SAS data set names?
- Can the data sets be opened successfully?

The remainder of this chapter describes how to add semantic information to the grammar.

Note: Chapter 21 provides complete reference information on all the grammar elements discussed in this chapter. Refer to Chapter 21 for more information on the arguments to semantic actions as well as other reference information.

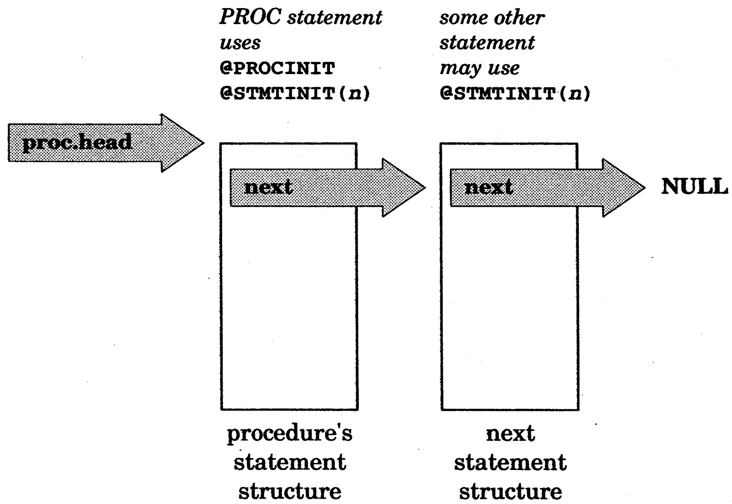
Passing Information from the User to the Procedure

Any time a user invokes a procedure, the procedure must be able to detect what arguments, statements, and values were used with the procedure. This information is passed to the procedure via a data structure called the *statement structure*. The grammar includes commands called *semantic actions*, which specify where to store each piece of information in the statement structure. Your procedure then accesses that data structure to determine how the procedure was invoked.

For example, some semantic actions indicate that a statement has been invoked and define a statement structure to contain information about the statement. Other semantic actions indicate that a data set name, a list of variables, a bit argument, or a parameter was specified when the procedure was invoked. In most cases, the information gathered by the semantic actions is stored in the procedure's statement structure. (It is also possible to create statement structures for each statement.)

Figure 3.1 illustrates how the @STMTINIT semantic action creates a statement structure each time it is invoked. You must use the @STMTINIT semantic action for the PROC statement. You can create other statement structures later in the grammar, but it is not necessary to create a separate statement structure for each statement in the procedure. This will be discussed in more detail later in this chapter. Where there are two or more statement structures, the `next` field points to the next structure in the chain.

Figure 3.1
Chained Statement Structures

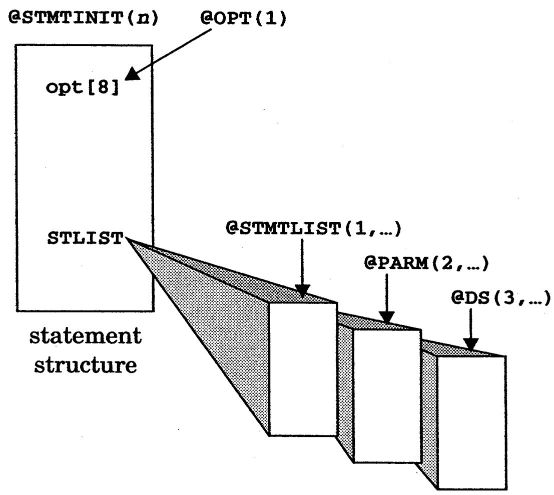


Once you have defined a statement structure, you can insert information into it with the following semantic actions:

- The @STMTLIST semantic action stores a variable list in a part of the first statement structure called a *statement field list* (STLIST). The statement field list is an array of structures that can store lists, parameter values, or data set names. In the following figure, the @STMTLIST action stores a variable list in the first statement field list of the statement structure.
- The @PARM and @DS semantic actions create other statement field lists to store the parameter value and the name of the SAS data set. In the following figure, the @PARM action stores a parameter's value in the second statement field list of the statement structure. The @DS semanatic action stores a data set name in the third statement field list.
- The @OPT semantic action sets a bit in an 8-byte field `opt[8]` to indicate that a bit argument was specified. This field permits up to 64 bit arguments to be associated with a statement. In this example, only the first bit in the field is being set.

Figure 3.2 illustrates the effect each of these semantic actions has on a statement structure.

Figure 3.2 *Storing Information in a Statement Structure*



There are many more semantic actions you can use with a grammar, but these handle the most common needs for your grammar. All the semantic actions are described in detail in Chapter 21.

Initializing the Procedure's Semantic Actions

The first two productions in the grammar do not require any semantic actions:

```
PROGRAM = ANYSTMT ENDJB ,

ANYSTMT = TESTITSTMT
        | VARSTMT
        | BYSTMT
        | COMPARESTMT ,
```

The TESTIT statement's production needs several important semantics. The @PROCINIT semantic is called once for a procedure, in the PROC statement. It signals the beginning of the procedure.

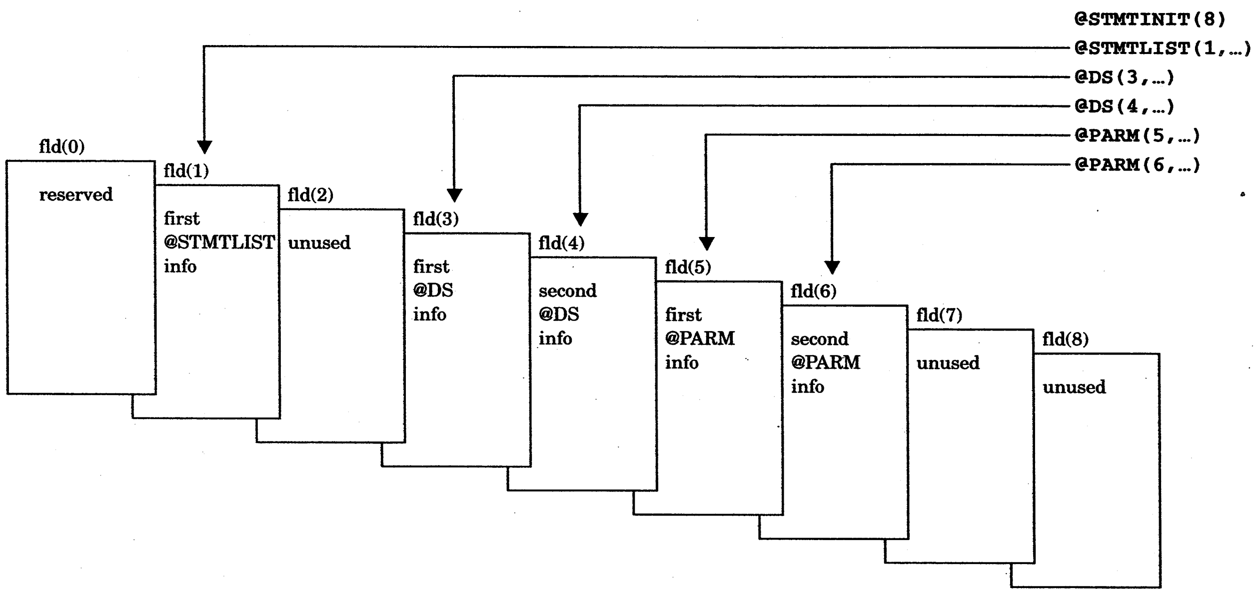
The next semantic action, @STMTINIT, creates the first statement structure. You must indicate in the @STMTINIT semantic action how much storage space needs to be allocated for the structure. That is, how many times will the grammar use semantic actions that store information in this particular statement structure? The primary semantic actions that store information in the statement structures are the following:

- @DS or @DSDFLT
- @PARM
- @STMTLIST.

In fact, the number you specify in the @STMTINIT semantic action sets up an array of storage areas, the `flds` array, for the data that are stored by the @DS, @PARM, and @STMTLIST semantic actions. When you use one of these actions, you specify which array element contains that data. You don't have to use all the array elements that you allocate with the @STMTINIT semantic action, but you cannot use a number for the first argument to @DS, @PARM, or @STMTLIST that is greater than the number specified in the @STMTINIT action.

Figure 3.3 illustrates how the semantic actions in this example allocate and use space in the first statement structure. Each of the actions is discussed in later sections. The @STMTINIT semantic action creates an array of eight available elements; three of these elements are not used in this example. Note that the first argument for the @DS, @PARM, and @STMTLIST semantic actions indicates which array element contains the information for that action.

Figure 3.3 *Creating and Using Storage Space in a Statement Structure*



And finally, the @STMTEND semantic action indicates that the definition is finished for this statement structure. This production, so far, looks like the following:

```
TESTITSTMT =                                  @PROCINIT
                                              @STMTINIT(8)
                "TESTIT"      TESTOPTS*
                                              @STMTEND ,
```

Note: A little later in the chapter you will see one more set of semantic actions added to this production, but this is a valid production even at this point.

Adding Semantic Actions to the VAR Statement

As you recall from the description of the procedure at the beginning of this chapter, the VAR statement permits a list of numeric variables. This section shows how to create a list of variables and ensure that the variables are numeric.

As mentioned earlier, not all statements in your procedure require a statement structure. You can use the procedure's statement structure to store lists of variables and other values instead of creating a separate statement structure. For example, in the grammar for the VAR statement, the variable list is stored in the procedure's statement structure by using the semantic action @STMTPROC before the @STMTLIST semantic action.

The @STMTLIST semantic action assigns a portion of the statement structure to store a list and establishes that list as the one currently being processed. The list is not created, however, until the grammar processor encounters one of the productions that describes a list, such as the VARLIST production.

The second argument for the @STMTLIST semantic action indicates the type of values or variables that are valid in this statement. The values permitted for the second argument are described in more detail in Chapter 21, but the more common ones are shown in Table 3.1.

Table 3.1 *Commonly Used Values for @STMTLIST*

Second Argument	Contents of List
1	numeric variables
2	character variables
3	mixed variables
4	numeric values (not variables)

In this case, the second argument is 1, so only numeric variables are permitted. This part of the semantic action permits you to validate the user's input with the grammar. The complete production for the VAR statement follows:

```
VARSTMT    =                                     @STMTPROC
                                                 @STMTLIST(1,1)
              ("VARIABLES" | "VAR")   VARLIST ,
```

The list of variables for the VAR statement is stored in the first statement list of the procedure's statement structure (remember that the first argument to @STMTLIST identifies the array element where the list is stored). If you plan to use the procedure's statement structure to store information for several statements, be sure to allocate enough space with the @STMTINIT action to create all the statement lists that you need.

Note: When you use a @STMTLIST action, the values that are stored in the statement structure are not actually the values of variables. Instead, the statement list contains the variable numbers from the data set. This will be discussed in more detail in Chapter 10, "Writing a SAS Procedure," and Chapter 20, "Data Structures."

Creating a Separate Statement Structure

Recall that the requirements for the COMPARE statement are that it names a valid numeric variable and then provides a list of numeric values.

COMPARE *variable=value-list*;

This statement differs from the VAR statement because it stores several types of information. For a statement like this, you might want to create a separate statement structure to store the information for that statement. Although this information can be stored in the procedure's statement structure, it is useful to group this information in a separate statement structure.

Use the @STMTINIT semantic action again to create the second statement structure. It is automatically chained to the procedure's statement structure through the `next` field. (Chapter 20 describes the fields of the statement structure in more detail.) Note that this statement structure allocates only two statement lists: one for the variable and one for the list of numeric values.

```
   #----------------- incomplete production --------------------#
COMPARESTMT =                                             @STMTINIT(2)
                "COMPARE"
                                        ONEVAR "="
                                        NUMLIST
```

To enable the user to specify a variable name and then a list of values, you must specify two @STMTLIST semantic actions to create two lists. The first list contains a single variable name (the ONEVAR production limits the list to a single element). Because you want the user to specify only the name of a numeric variable, use 1 for the second argument. Thus, to achieve the first half of the COMPARE statement, use this grammar coding:

```
   #----------------- incomplete production --------------------#
COMPARESTMT =                                             @STMTINIT(2)
                "COMPARE"                                 @STMTLIST(1,1)
                                        ONEVAR "="
```

Finally, the last part of the COMPARE statement requires a list of numeric values, not variables. The NUMLIST production, described earlier, indicates that the user can specify a list of numeric values. Now add the @STMTLIST semantic action to store these values, but unlike earlier uses of this action, this time specify 4 for the second argument. A value of 4 indicates that the data to be stored in the list are numeric values, not variables.

The complete production follows. As in the example for the TESTITSTMT production, the @STMTEND action signals the end of the definition of this statement structure.

```
COMPARESTMT =                                             @STMTINIT(2)
                "COMPARE"                                 @STMTLIST(1,1)
                                        ONEVAR "="        @STMTLIST(2,4)
                                        NUMLIST           @STMTEND,
```

Defining Semantic Actions for Arguments

To complete the semantic information for the procedure's statement, you still need to define the arguments that appear in the PROC TESTIT statement. This procedure includes the three primary types of arguments: parameters, bit arguments, and data set names.

Note: This chapter does not attempt to describe all the possible values for arguments to the semantic actions discussed here. Refer to Chapter 21 for detailed information on the arguments for semantic actions.

Data Sets

To indicate that a SAS data set name should be stored in the statement structure, use the @DS semantic action. The first three arguments to the @DS semantic action indicate the following:

- the statement list where the data set name is stored in the statement structure.
- the way the data set is being used. A 1 indicates input; a 2 is for output. Other modes are described in Chapter 21.
- whether the data set is the lookup data set for variable lists. A 1 indicates the data set is the lookup data set; 0 means it is not. A *lookup data set* is the data set the parser uses to validate the list of variables specified in a statement. For example, when you specify variables in the VAR, BY, or ID statement of the PRINT procedure, those variables must exist in the data set specified in the DATA= option because that is specified as the lookup data set in the grammar for PROC PRINT.

With this information, you can begin building the semantic actions for the input and output data sets as shown here:

```
TESTOPTS    = ("DATA"          "="   DSFIELD      @DS(3,1,1,4,2,4,0))
            | ("OUT"           "="   DSFIELD      @DS(4,2,0,2,2,4,0))
```

The first three arguments to the @DS semantic actions in this example indicate that the DATA= data set is stored in the third statement list, its mode is input, and this is the *lookup data set.* The lookup data set is the data set used to verify that any variables listed in a statement are existing variables in a SAS data set. The OUT= data set reference is stored in the fourth statement list, the mode of the data set is output, and this is not the lookup data set.

The next three arguments to the @DS semantic action indicate the type of access the procedure needs to have for the data set. These three arguments work together and are set depending on the way in which your procedure uses a data set. For a simple input data set that is read once sequentially (with or without BY groups), specify 4,2,4 for the these arguments. For an output data set, specify 2,2,4.

The seventh argument indicates whether the procedure should continue processing when an I/O error occurs. In most cases, your procedures should end when an I/O error occurs, so specify 0.

Note: If your procedure performs BY processing, you must set this argument to 0.

An optional eighth argument indicates when a data set cannot have a WHERE statement or attribute statement (such as ATTRIB, FORMAT, INFORMAT, or LABEL) applied to it. In most cases, you omit this argument and the WHERE and attribute statement will automatically work with your procedure.

The complete semantic actions for the two data sets are shown in the following example. Keep in mind that this is only part of the semantic actions for TESTOPTS production.

```
    # incomplete production #
TESTOPTS     = ("DATA"             "="   DSFIELD       @DS(3,1,1,4,3,4,0))
             | ("OUT"              "="   DSFIELD       @DS(4,2,0,2,2,4,0))
             |(("HIVAL" | "HI")    "="
                                         INT)
             |(("LOVAL" | "LO")    "="
                                         INT)
             | ("YEARLY"
             | ("MONTHLY"
```

Parameters

Storing the values of parameters is somewhat easier than defining data sets. For storing parameter values, use the @PARM semantic action. Like other semantic actions, the first argument to the @PARM action specifies which statement list contains the data. The second argument indicates the type of data to be stored. For this example, both the HIVAL and LOVAL values are numeric, so the second argument is 1. Refer to Chapter 21 for a complete list of possible values for the second argument of the @PARM semantic action.

The TESTOPTS production now looks like this:

```
    # incomplete production #
TESTOPTS     = ("DATA"             "="   DSFIELD       @DS(3,1,1,4,3,4,0))
             | ("OUT"              "="   DSFIELD       @DS(4,2,0,2,2,4,0))
             |(("HIVAL" | "HI")    "="                 @PARM(5,1)
                                         INT)
             |(("LOVAL" | "LO")    "="                 @PARM(6,1)
                                         INT)
             | ("YEARLY"                                         )
             | ("MONTHLY"                                        )
```

Bit Arguments

The final addition to the TESTOPTS production is to specify the semantic actions for the two bit arguments: YEARLY and MONTHLY. Use the @OPT semantic action to set a bit in the `opt` field of the statement structure. The two bit arguments in this procedure set the first two bits of the field. The complete production for TESTOPTS follows:

```
TESTOPTS     = ("DATA"              "="   DSFIELD      @DS(3,1,1,4,3,4,0))
             | ("OUT"               "="   DSFIELD      @DS(4,2,0,4,3,4,0))
             |(("HIVAL" | "HI")     "="                @PARM(5,1)
                                          INT)
             |(("LOVAL" | "LO")     "="                @PARM(6,1)
                                          INT)
             | ("YEARLY"                               @OPT(1))
             | ("MONTHLY"                              @OPT(2)) .
```

Remember that the period follows this production because it is the last production in the grammar.

Adding Default Data Sets

The grammar for the TESTIT procedure is complete except for one capability: both the DATA= and OUT= data set specifications are optional. The procedure should use the default data sets, _LAST_ and _DATA_, when the user does not specify these options. You must add another semantic action to the procedure's production to indicate that default data sets are to be used when no data set is specified.

The @DSDFLT semantic action indicates that a default data set is to be opened. If the user does not specify DATA= in the PROC TESTIT statement, the _LAST_ data set is opened. Likewise, if OUT= is not specified, _DATA_ is opened. If you do not want to open a data set unless one is explicitly specified, omit the @DSDFLT action.

The @DSDFLT semantic action has the same arguments as the @DS semantic action, so you can copy the same values that you specified in the TESTOPTS production. The revised TESTITSTMT production follows:

```
TESTITSTMT =                                    @PROCINIT
                                                @STMTINIT(8)
                "TESTIT"      TESTOPTS*         @DSDFLT(3,1,1,4,3,4,0)
                                                @DSDFLT(4,2,0,2,2,4,0)
                                                @STMTEND ,
```

Complete Example

The final grammar, complete with comments, is illustrated here.

```
#------------------------------ Sample Grammar -----------------------#

%INCLUDE "STUBGRM"
PROGRAM = ANYSTMT ENDJB ,

ANYSTMT = TESTITSTMT
        | VARSTMT
        | BYSTMT
        | COMPARESTMT ,
```

```
#------------------------- Statement Productions --------------------#

TESTITSTMT =                                        @PROCINIT
                                                    @STMTINIT(8)
              "TESTIT"      TESTOPTS*               @DSDFLT(3,1,1,4,3,4,0)
                                                    @DSDFLT(4,2,0,4,3,4,0)
                                                    @STMTEND ,

VARSTMT    =                                        @STMTPROC
                                                    @STMTLIST(1,1)
             ("VARIABLES" | "VAR")   VARLIST ,

COMPARESTMT =                                       @STMTINIT(2)
              "COMPARE"                             @STMTLIST(1,1)
                                     ONEVAR "="     @STMTLIST(2,5)
                                     NUMLIST        @STMTEND,

#------------------------ PROC Statement Options --------------------#

TESTOPTS     = ("DATA"          "="  DSFIELD        @DS(3,1,1,4,3,4,0))
             | ("OUT"           "="  DSFIELD        @DS(4,2,0,4,3,4,0))
             |(("HIVAL" | "HI") "="                 @PARM(5,1)
                                     INT)
             |(("LOVAL" | "LO") "="                 @PARM(6,1)
                                     INT)
             | ("YEARLY"                            @OPT(1))
             | ("MONTHLY"                           @OPT(2)) .

#--------------------------- End of Grammar ------------------------#
```

Chapter 4 Creating the Grammar Function

Introduction

This chapter describes how to use the USERPROC procedure with a grammar source file and what the USERPROC procedure creates. In addition, this chapter explains how your procedure accesses the grammar function created by PROC USERPROC. For detailed reference information on the USERPROC procedure, refer to Chapter 19, "The USERPROC Procedure."

The USERPROC Procedure and Grammars

Before your procedure can use the grammar you have created, you must process the grammar source code with the USERPROC procedure. PROC USERPROC creates a grammar function in the same language you are using for your procedure: C, FORTRAN, PL/I, or IBM 370 assembler. Your procedure can then call the grammar function to parse the user's statements. Figure 5.1 in Chapter 5, "User-Written Procedures and the SAS System," illustrates the relationship between the grammar source code, the grammar function, and your procedure.

When you use PROC USERPROC to process a grammar source file, you must specify these arguments:

- The FUNCFILE= argument specifies the location of the grammar function created by the USERPROC procedure. Specify this argument in one of the following forms:

  ```
  FUNCFILE=fileref
  FUNCFILE='external-file'
  ```

  This file contains source code generated by PROC USERPROC in the language that you specify (which must be the same language you are using to write the main procedure). You can specify either a fileref or the file specification for the file in the aggregate storage location. If you use a file specification on MVS, this argument names a sequential file or a partitioned data set and member, for example MYLIB.TOOLKIT.SRC(SAMPLE). On CMS, the file specification names the filename, filetype, and, optionally, the filemode, for example, SAMPLE TEXT A. On directory-based systems, the file specification provides the pathname for the file, for example, [MYLIB.TOOLKIT]SAMPLE.

- The GRAMFUNC= argument names the grammar function. Specify this argument as follows:

  ```
  GRAMFUNC=function-name
  ```

  To simplify naming conventions, you can use the same name for the GRAMFUNC= argument as the member name (for MVS), the filename (for CMS), or the name of the file in the directory (on directory-based systems) in the FUNCFILE= argument. For example, using the files described earlier for the FUNCFILE= argument, you could specify SAMPLE for the GRAMFUNC= argument. It is not necessary, however, that the names match. Keep in mind that the name you provide for this argument is the name that your procedure uses to call the grammar function.

- The GRAMMAR= argument specifies the location of the file that contains the grammar statements you coded. Specify this argument in one of the following forms:

  ```
  GRAMMAR=fileref
  GRAMMAR='external-file'
  ```

In addition to these required arguments, several optional arguments are available. Use the GRAMLANG= option to specify the language in which the grammar function should be created. If you omit this option, the grammar function is created in C. Use the INCLLIB= option to specify the libraries containing files that you include in your grammar with a %INCLUDE statement.

Sample PROC USERPROC Step

The following example illustrates how to invoke PROC USERPROC for the TESTIT grammar developed in Chapter 3, "Writing a Grammar." In this example, the grammar function is being created for a C procedure. This example also uses the INCLLIB= option to show how to access portions of a grammar that are used by your grammar but stored in other files in an aggregate storage location (a directory, partitioned data set, or TXTLIB).

```
filename mygrams 'file specification';
filename mycsrc  'file specification';
filename moregram 'aggregate storage location';

proc userproc grammar=mygrams funcfile=mycsrc
              gramfunc=testitg gramlang=c incllib=moregram;
```

When you write your procedure, you call the grammar function in the call to SAS_XSPARSE, which invokes the parser. Be sure that you declare the grammar function before using it. This is how the generated grammar function is used in a C procedure:

```
ptr testitg U_PARMS((void));

SAS_XSPARSE(testitg(), NULL, &proc);
if (proc.error >= XEXITPROBLEM)
   SAS_XEXIT(XEXITSYNTAX, 0);
```

Part 3 of this book, "Procedures," describes how to write procedures and use the grammar function.

Part 3
Procedures

Chapter 5 User-Written Procedures and the SAS® System

Introduction

This chapter expands on the information provided in Chapter 2, "Understanding Grammars," by providing more information on how your procedure works with the SAS System. This chapter describes how to perform each common programming task so that your program interfaces with and fully uses the capabilities of the SAS System. Examples of how to perform these tasks are given in Chapter 10, "Writing a SAS Procedure."

This chapter also briefly discusses the languages you can use to write a SAS procedure.

Accessing the Power of the SAS System

When you write a SAS procedure using SAS/TOOLKIT software, you need to perform special functions not needed in other programs you write, such as working with SAS data sets and parsing the user's statements. To simplify these tasks, you use the SAS_X and SAS_Z routines, which are part of SAS/TOOLKIT software. The following sections explain the special tasks your procedure performs using SAS/TOOLKIT software.

Obtaining Information Supplied by the User

Chapter 2 introduces the role of the grammar in communicating to your procedure. As you may recall from Chapter 2, the parser interprets statements entered by the user and determines what actions to take. As the parser uses your grammar to validate the user's statements, it also stores the information from those statements in a series of data structures called the *statement structures*.

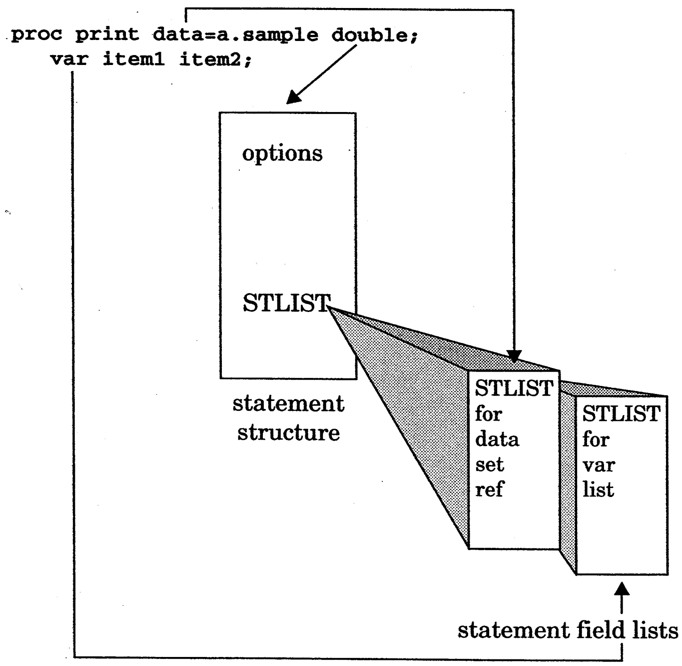
Before your procedure can run, it must determine what information the user specified in the statements that invoked the procedure.

The parser stores the following kinds of information in the statement structures:

- the fileid of the input data set
- the statements the user specified to invoke the procedure
- the arguments specified in each statement
- the variables the user wants to process.

Different types of information are stored in different portions of the statement structure. Figure 5.1 illustrates where information from the user's statements is stored in the statement structures.

Figure 5.1
User's Information Stored in Statement Structures



The statement structure is described in greater detail in Chapter 20, "Data Structures." This structure is a complex storage mechanism, but accessing the structure is greatly simplified when you use the SF routines. Chapter 10 illustrates how to call the SF routines to extract information from the statement structures.

Opening and Closing SAS Data Sets

When you write a SAS procedure, you do not use standard programming commands to open and close files when you're working with SAS data sets. In most cases, if you have defined a data set reference in the grammar so that the user can specify which SAS data set to use for input or output (that is, if you have a DATA= or OUT= option defined in your grammar), the data set is opened during the parsing process. In addition, when your procedure ends, the data set is automatically closed by the SAS Supervisor. Thus, your procedure does not have to call any routines to open or close SAS data sets.

If you want to open or close a data set from within the procedure, use the SAS_XOOPEN and SAS_XOCLOSE routines.

Reading and Writing Observations

To read or write an observation from a SAS data set, you must call the SAS_XV routines. You cannot use calls to standard programming commands or functions to read observations. Reading observations requires several steps:

1. Determine which variables the user wants to read.
2. Describe to the SAS System which variables to read from each observation and indicate where to store those variables in the program.
3. Read the observations, one at a time, and process them.

Writing observations requires these steps:

1. Determine which variables the user wants to write.
2. Create new variables, if needed.
3. Describe to the SAS System which variables to write to the output data set for each observation and indicate where those variables are stored by the program.
4. Write each observation.

Accessing External Files

You can access external files, that is non-SAS files, from a SAS procedure by simply using the programming statements or functions for file I/O that are provided with the programming language you are using. SAS/TOOLKIT software does not include any special routines for reading from or writing to external files.

Printing Output

Although it is possible to write output from a SAS procedure using standard programming commands or functions, the recommended destination for output from user-written procedures is the SAS log and procedure output file. Writing messages to the log and reports to the procedure output file causes your output to be integrated with other SAS output. Thus, your output is automatically displayed by the SAS Display Manager System if that is how the user runs the procedure. In addition, writing to these locations enables users to more easily manage the output from your program. Use the SAS_XP routines to write output to the SAS log and procedure output files.

Error Processing

SAS/TOOLKIT software provides extensive error information to enable you to check results of function calls and detect errors as you interface with the SAS System. Most of the functions included in the library of SAS_X and SAS_Z routines return a code indicating the success of the function call. Use these return codes to detect errors and then use the SAS_XEXIT to exit appropriately. Many error messages are available to describe error conditions that occur. Use the SAS_XEXIT and SAS_XPRLOG routines to issue messages based on return codes returned by function calls.

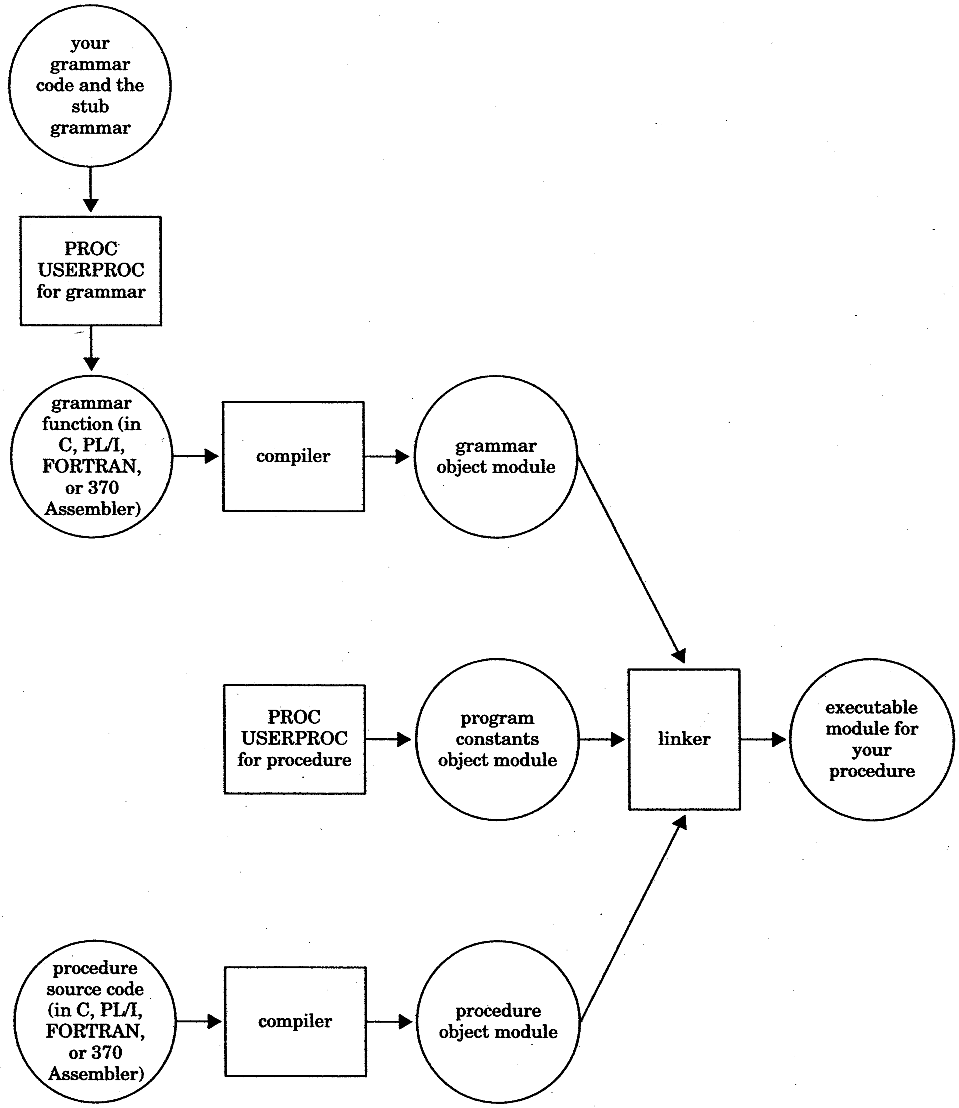
Preparing the Procedure for Use

For both the procedure you write and the grammar that you code, you must run a special SAS/TOOLKIT procedure, PROC USERPROC. The USERPROC procedure serves several functions. When you use it with a grammar, it creates a function in the same language that you have used to write your procedure. You compile this function and then link it to the compiled module for your procedure.

When you use PROC USERPROC for a procedure, it creates a program constants object module, which must be linked to the procedure to enable you to use your procedure with the SAS System.

Figure 5.2 illustrates the process of writing the grammar code, writing the procedure code, running PROC USERPROC, and linking all modules into an executable module. The executable module is the program that can be invoked by the user from within a SAS session by using the PROC statement.

Figure 5.2
Steps for Creating a User-Written Procedure



Choosing a Language for Writing Procedures

SAS/TOOLKIT software allows you to use C, FORTRAN, PL/I, or IBM 370 assembler language to write a SAS procedure. Table 5.1 lists the supported languages for each operating system.

Table 5.1 *Supported Languages for CMS, MVS, and VMS Operating Systems*

Host Operating System	Language	Supported Version
CMS	C	SAS/C Compiler
	FORTRAN	VS FORTRAN Version 2 compiler
	PL/I	Version 2, PL/I Optimizing compiler
	assembler	IBM 370 Assembler
MVS	C	SAS/C Compiler
	FORTRAN	VS FORTRAN Version 2 compiler
	PL/I	Version 2, PL/I Optimizing compiler
	assembler	IBM 370 Assembler
VMS	C	native VMS C compiler
	FORTRAN	native VMS FORTRAN compiler
	PL/I	native VMS PL/I compiler

Differences in Languages

You can use any of the supported languages to write your procedure. SAS software is written primarily in the C language, but SAS/TOOLKIT software enables you to work in the language of your choice. Keep in mind that the supported languages have some fundamental differences that you should understand when you work in a language other than C.

In particular, C is a *call-by-value* language, which means that when you call a function, the actual values of the function's arguments are passed to the function. The practical implication of this language feature is that you cannot directly change the value of an argument. Both PL/I and FORTRAN are *call-by-address* languages, which means the addresses of the values for a function's arguments are passed to the function. For these languages, you can change the value of an argument to a function. With SAS/TOOLKIT software, IBM 370 assembler is treated as a call-by-value language.

The differences between call-by-value and call-by-address languages may cause you to encounter some unfamiliar coding techniques. For example, if a function needs to update one of the arguments to the function, you must pass the address of the variable to be updated. Although this would not be necessary in standard FORTRAN or PL/I programs, your SAS procedure must use this technique to interface correctly with SAS/TOOLKIT software.

In addition to differences between languages, you may encounter differences between different compilers of the same language. For recommended coding techniques and information on writing portable programs, refer to Chapter 25, "Writing Programs in the C Language," Chapter 26, "Writing Programs in the FORTRAN Language," and Chapter 27, "Writing Programs in the PL/I Language." Note that portability is not an issue for assembler language since IBM 370 assembler is portable between CMS and MVS operating systems and VMS assembler is not supported.

Using More than One Language

It is possible to write your procedure using more than one language, but you must be sure that all calls to SAS/TOOLKIT routines are made in the same language. SAS/TOOLKIT software establishes the language environment as the procedure begins executing. After determining what language is being used, SAS/TOOLKIT software handles argument lists and other language-dependent issues based on the language.

▶ *Caution* .... ***If you attempt to call SAS/TOOLKIT routines using two languages, argument lists will not be handled appropriately and unexpected results will occur.*** ▲

Chapter 6 Building a Template for SAS/TOOLKIT™ Procedures in the C Language

Introduction

This chapter introduces the most basic steps to writing a SAS procedure using the C language. When you finish reading this chapter, skip to Chapter 10, "Writing a SAS Procedure," for the remaining steps in building your procedure.

Setting Up a Template

To set up a template for a SAS procedure written in C, you need to include the following steps:

- defining modules
- accessing SAS/TOOLKIT header files
- naming the grammar function and the executable module for your procedure
- setting up the SAS/TOOLKIT environment
- calling the parser and checking for errors
- calling other SAS/TOOLKIT routines
- ending a user-written procedure.

These steps are explained in detail in the remainder of this chapter.

Defining Modules

When you write a SAS procedure in C, you can break your procedure into multiple source code files, as you can with any other C program. All source code modules for SAS/TOOLKIT procedures need to include a **#define** statement to indicate that the module is part of a SAS/TOOLKIT procedure. In the main procedure source file, include the following line to indicate that this module is the main procedure source file:

```
#define MAINPROC 1
```

Do not include this statement in any other source file.

All source code files must include the following statement to indicate that the module is part of a SAS procedure:

```
#define SASPROC 1
```

In the main procedure source file, the SASPROC definition follows the MAINPROC definition; in all other source code files, SASPROC is used alone.

The SASPROC symbol signals SAS/TOOLKIT software to include definitions needed for procedure-writing. Note that when you write an IFFC, you use a similar symbol that includes definitions that are appropriate for writing IFFCs instead of procedures.

Accessing SAS/TOOLKIT Header Files

The next step is to include the SAS/TOOLKIT header files. Use this statement to access all the necessary definitions for writing procedures in C:

```
#include <uwproc>
```

The **uwproc** include file contains many **#define** and **typedef** definitions and also includes other host-specific include files. All of the special data types used in the examples in this book, such as **ptr**, **rctype**, and the various structures, are defined in the **uwproc** file or the files it includes.

Note: Be sure to use the **<uwproc>** format for the **#include** statement, not **"uwproc.h"**. Angle brackets are valid on all systems that run SAS/TOOLKIT software.

After including **uwproc**, you can include any other header files or **#define** statements needed for your program.

Naming the Grammar Function and the Executable Module for Your Procedure

Before you begin the main function of your program, you need to declare both the grammar function and the executable module for your procedure. To declare the correct type returned by the grammar function, include this line in your program:

```
ptr grammar-function U_PARMS((void));
```

where *grammar-function* is the output of the grammar processor and is the same name as the value you specify for the GRAMFUNC= option in the PROC USERPROC statement for your grammar.

The next step is to name the executable module for your procedure. The executable module is the result of compiling and linking your procedure. You name the executable module in the declaration for the main function in your program. Specify the following line after the declaration for the grammar function:

```
void U_MAIN(exec-name)()
```

where *exec-name* is the name of the executable module. This name must be the same name as the name used to invoke your procedure. For example, if your procedure is called ABCDEF, the U_MAIN statement is as follows:

```
void U_MAIN(abcdef)()
```

At this point, you can declare any necessary static and automatic variables for the main function of your program.

Setting Up the SAS/TOOLKIT Environment

The first executable statement in the main procedure code must be the following:

```
UWPRCC(&proc);
```

The `proc` pointer is defined by the `uwproc` include file. The UWPRCC routine is the SAS/TOOLKIT routine that sets up the interface between your C procedure and the SAS/TOOLKIT environment.

▶ *Caution* .... ***You must call the UWPRCC routine for your procedure to work correctly.*** ▲

Calling the Parser and Checking for Errors

Before you can access the information specified by the user in the statements that call your procedure, you must parse the statements. To parse the statements, use the SAS_XSPARSE routine:

```
SAS_XSPARSE(grammar-function(), NULL, &proc);
```

The *grammar-function* specified in the call to SAS_XSPARSE must match the *grammar-function* specified in the U_PARMS declaration, described earlier in this chapter. The second argument is always NULL, and the third argument is always `&proc`.

Immediately after you parse the statements, check for parsing errors, as in this example:

```
if (proc.error >= XEXITPROBLEM)
   SAS_XEXIT(XEXITSYNTAX, 0);
```

The XEXITPROBLEM and XEXITSYNTAX symbols (as well as many others) are defined in the `uwproc` include file. When you exit the procedure and specify one of the XEXIT*xxxx* symbols, an appropriate error message is printed by the SAS System. Refer to the description of the SAS_XEXIT routine in Chapter 22, "SAS_X Routines," for more information on exit codes.

Calling Other SAS/TOOLKIT Routines

The main action of your program is a combination of calls to various SAS_X routines, SAS_Z routines, and standard C programming statements. In general, you use SAS_X or SAS_Z routines any time you perform I/O, allocate memory, manage character strings, format data, or exit. In addition, the SAS_Z routines provide special routines for mathematical and statistical functions and for handling missing values.

Chapter 10 explains the primary routines needed in most SAS procedures. Refer to Chapter 22 for detailed reference information on the SAS_X routines and to Chapter 23, "SAS_Z Routines," for reference information on those routines.

Ending a User-Written Procedure

When you are ready to terminate the procedure, call SAS_XEXIT.

▶ *Caution* .... ***Do not use the return statement to end the U_MAIN function in your procedure. Never use the stop statement.*** ▲

Call SAS_XEXIT as follows:

```
SAS_XEXIT(XEXITNORMAL, 0);
```

The XEXITNORMAL symbol, like the XEXITPROBLEM and XEXITSYNTAX symbols discussed earlier in this chapter, is defined in the `uwproc` include file. Refer to the description of the SAS_XEXIT routine in Chapter 22 for more information on exit codes.

Summary

This section summarizes the template for a SAS/TOOLKIT procedure written in C.

```
#define MAINPROC 1      /* use in the main source only              */
#define SASPROC 1       /* required for all source modules          */

#include <uwproc>       /* required in the main source module       */
other #include statements

ptr grammar-name U_PARMS((void)); /* grammar function prototype     */
void U_MAIN(exec-name)()     /* main proc only - use executable name*/
   {
   declarations, as needed

   UWPRCC(&proc);       /* first executable statement in main proc  */
   SAS_XSPARSE(grammar-name(), NULL, &proc);     /* parse statements      */
   if (proc.error >= XEXITPROBLEM)  /* check for successful parsing */
      SAS_XEXIT(XEXITSYNTAX, 0);
   main procedure code here
   SAS_XEXIT(XEXITNORMAL, 0);       /* do not do a return or stop   */
}
```

Before you write the main procedure code, it's a good idea to build your grammar and the template shown above, run the USERPROC procedure, compile both the grammar and the procedure, and then test the skeletal procedure to be sure no errors occur. After you understand the grammar, basic procedure template, and the process of running PROC USERPROC, compiling, and linking, you can expand your procedure to its full functionality.

You may also want to keep this template available as a starting point for procedures you will develop later.

Chapter 7 Building a Template for SAS/TOOLKIT™ Procedures in the FORTRAN Language

Introduction

This chapter introduces the most basic steps to writing a SAS procedure using the FORTRAN language. In addition, it explains briefly an extra step in the compiling and linking process that enables you to write FORTRAN code using special symbolic names and other features not commonly available in FORTRAN programs. When you finish reading this chapter, skip to Chapter 10, "Writing a SAS Procedure," for the remaining steps in building your procedure.

Preprocessing FORTRAN Programs

The FORTRAN language has several restrictions that require special handling by SAS/TOOLKIT software to enable a FORTRAN procedure to interface with the SAS System. This section explains how SAS/TOOLKIT software overcomes some restrictions of the FORTRAN language.

Note: This section is for your information only. You are not required to code your FORTRAN procedure in any special manner to overcome these restrictions; all of the problems are handled by SAS/TOOLKIT software.

When you compile and link your SAS procedure, you include an extra step that invokes a preprocessor developed for SAS/TOOLKIT software. This preprocessor is a SAS program, FTNPREP, that edits your program and changes your program to conform to the requirements of the compiler that you are using. The process for compiling and linking is described in detail in the appendix for your operating system. The following sections describe the kinds of changes that FTNPREP makes to your program before it is compiled.

Include Files

The implementation of FORTRAN differs widely under different operating systems. Some compilers permit you to include other files into your FORTRAN program and others do not. In addition, the syntax of the INCLUDE statement (on the compilers that support it) is different. The FORTRAN preprocessing program included with SAS/TOOLKIT software, FTNPREP, overcomes these differences as follows:

- The preprocessor determines the operating system running the job, and converts the INCLUDE statements to the appropriate syntax for that operating system.
- If the INCLUDE statement is not supported in any form for the FORTRAN compiler on an operating system, the preprocessor automatically copies the member to be included into the source file.

Symbolic Names

All SAS/TOOLKIT routines have symbolic names. Many routines are prefixed with either SAS_X or SAS_Z, and the rest of the name is a mnemonic name that refers to the action of the routine. The mnemonic name is replaced by the actual routine name, in the form SAS*nnn*. FORTRAN does not have any kind of symbolic substitution. SAS/TOOLKIT software uses the preprocessor to translate the mnemonic names to the actual routine names. Therefore, you can use the mnemonic routine names described in this book to write a more understandable program. In addition, the preprocessor substitutes constants for other symbols, such as the XEXIT codes.

Coding for Host Operating System Dependencies

The preprocessor program also provides the ability to add code to your program that is operational only on a certain operating system. To use this feature, place a three-character string in columns 2 through 4 to indicate the operating system that can process the line. The format for this statement is shown here with column indicators to show the correct placement on a line. Note that the statement begins in column 7 or later.

```
----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8
 VMS  VMS-dependent statement
```

The preprocessor looks for nonnumeric data in columns 2 through 4 of lines other than comments. If character data are found, the preprocessor compares the string against the first three characters of the operating system name, MVS, CMS, or VMS. In the previous example, if the operating system where the preprocessor is running is VMS, the line is kept in the source file, but the 'VMS' string is changed to blanks. If the operating system is not VMS, the line is dropped from the source file.

Setting Up a Template

To set up a template for a SAS procedure written in FORTRAN, you need to include the following steps:

- defining modules
- accessing necessary SAS/TOOLKIT files
- declaring the grammar function
- setting up the SAS/TOOLKIT environment
- calling the parser and checking for errors
- calling other SAS/TOOLKIT routines
- ending a user-written procedure.

These steps are explained in detail in the remainder of this chapter.

Defining Modules

When you write a SAS procedure in FORTRAN, you can break your procedure into multiple source code files, as you can with any other FORTRAN program. At the beginning of the main module, include this line in all SAS procedures that you write, even if you are currently not running under VMS. On CMS and MVS, this line is ignored and the subroutine name is MAIN. On VMS, the name needs to be the name of the executable module.

```
VMS  SUBROUTINE name-of-executable
```

where *name-of-executable* is the same name you specify in the MODNAME= option in the PROC USERPROC statement.

Note: Remember to start this line in column 2.

Accessing SAS/TOOLKIT Header Files

The next step is to include the SAS/TOOLKIT include file and other necessary include files.

Note: Use the INCLUDE statements shown here even if your compiler does not support the INCLUDE statement or has a different syntax for the INCLUDE statement. The preprocessor provided with SAS/TOOLKIT software ensures that the INCLUDE statement is handled correctly for each operating system.

Use the following statements to access necessary definitions for writing procedures in FORTRAN:

```
INCLUDE (PROTOSF)
INCLUDE (PROC)
```

The PROTOSF file contains the declarations for all of the SAS/TOOLKIT routines. The PROC file contains the procedure parsing structure, which is required in all procedures.

After these statements, you can include any other files needed for your program. In particular, if you are using any of the structures described in Chapter 20, "Data Structures," you must include the definitions of those structures. Use any of the following statements that apply:

```
INCLUDE (FATTRSTR)
INCLUDE (MEMLIST)
INCLUDE (NAMESTR)
INCLUDE (WZENV)
INCLUDE (XCOPSTR)
INCLUDE (XOOPNSTR)
```

Note: Each of these INCLUDE statements defines a COMMON block. You must be sure to include the member in each routine that references the COMMON block.

Declaring the Grammar Function

Before you begin the main task of your program, you need to declare the grammar function as follows:

```
INTEGER*4 grammar-function
```

where *grammar-function* is the output of the grammar processor and is the same name as the value you specify for the GRAMFUNC= option in the PROC USERPROC statement for your grammar.

Setting Up the SAS/TOOLKIT Environment

The next executable statement in the main procedure code must be the following:

```
CALL UWPRCF(IADDR(PROCHEAD))
```

The PROCHEAD field is defined in the PROC COMMON block, which you have already included. The UWPRCF routine is the SAS/TOOLKIT routine that sets up the interface between your FORTRAN procedure and the SAS/TOOLKIT environment. Note the use of the IADDR routine to obtain the address of the PROCHEAD field. This is a special routine provided with SAS/TOOLKIT software to create needed pointer types to use in writing SAS procedures. Refer to the description of IADDR in Chapter 24, "Miscellaneous Routines and Values."

▶ ***Caution*** .... ***You must call the UWPRCF routine for your procedure to work correctly.*** ▲

Calling the Parser and Checking for Errors

Before you can access the information specified by the user in the statements that call your procedure, you must parse the statements. To parse the statements, use the SAS_XSPARSE routine and the grammar function that you declared in an earlier step.

```
CALL SAS_XSPARSE(grammar-function(0),0,IADDR(PROCHEAD))
```

The second argument to SAS_XSPARSE is always 0, and the third argument is always the address of the PROCHEAD field.

Immediately after you parse the statements, check for parsing errors, as in this example:

```
 IF (PROC.ERROR .GE. XEXITPROBLEM)
+   CALL SAS_XEXIT(XEXITSYNTAX,0)
```

The XEXITPROBLEM and XEXITSYNTAX symbols (as well as many others) are handled by the preprocessor. When you exit the procedure and specify one of the XEXIT*xxxx* symbols, an appropriate error message is printed by the SAS System. Refer to the description of the SAS_XEXIT routine in Chapter 22, "SAS_X Routines," for more information on exit codes.

Calling Other SAS/TOOLKIT Routines

The main action of your program is a combination of calls to various SAS_X routines, SAS_Z routines, and standard FORTRAN programming statements. In general, you use SAS_X or SAS_Z routines any time you perform I/O, allocate memory, manage character strings, format data, or exit. In addition, the SAS_Z routines provide special routines for mathematical and statistical functions and for handling missing values.

Chapter 10 explains the primary routines needed in most SAS procedures. Refer to Chapter 22 for detailed reference information on the SAS_X routines and to Chapter 23, "SAS_Z Routines," for reference information on those routines.

Ending a User-Written Procedure

When you are ready to terminate the procedure, call SAS_XEXIT.

▶ ***Caution*** .... ***Do not use the RETURN statement to end the main procedure. Never use the STOP statement.*** ▲

Call SAS_XEXIT as follows:

```
CALL SAS_XEXIT(XEXITNORMAL,0)
```

The XEXITNORMAL symbol, like the XEXITPROBLEM and XEXITSYNTAX symbols discussed earlier in this chapter, is defined in the UWPROC include file. Refer to the description of the SAS_XEXIT routine in Chapter 22 for more information on exit codes.

Summary

This section summarizes the template for a SAS/TOOLKIT procedure written in FORTRAN.

```
C ****USE THE FOLLOWING THREE LINES IN ALL PROCEDURES
 VMS  SUBROUTINE name-of-executable
      INCLUDE (PROTOSF)
      INCLUDE (PROC)

      INTEGER*4 grammar-function

C ****THIS MUST BE THE FIRST EXECUTABLE STATEMENT
      CALL UWPRCF(IADDR(PROCHEAD))

C ****PARSE THE STATEMENTS
      RC = SAS_XSPARSE(grammar-function(0),0,IADDR(PROCHEAD))
      IF (PROCERROR .GE. XEXITPROBLEM)
     +   CALL SAS_XEXIT(XEXITSYNTAX,0)

C ****DO NOT USE STOP OR RETURN TO TERMINATE THE PROCEDURE
      CALL SAS_XEXIT(XEXITNORMAL,0)
      END
```

Before you write the main procedure code, it's a good idea to build your grammar and the template shown above, run the USERPROC procedure, compile both the grammar and the procedure, and then test the skeletal procedure to be sure no errors occur. After you understand the grammar, basic procedure template, and the process of running PROC USERPROC, compiling, and linking, you can expand your procedure to its full functionality.

You may also want to keep this template available as a starting point for procedures you will develop later.

Chapter 8 Building a Template for SAS/TOOLKIT™ Procedures in the PL/I Language

Introduction

This chapter introduces the most basic steps to writing a SAS procedure using the PL/I language. When you finish reading this chapter, skip to Chapter 10, "Writing a SAS Procedure," for the remaining steps in building your procedure.

Setting Up a Template

To set up a template for a SAS procedure written in PL/I, you need to include the following steps:

- defining modules
- accessing SAS/TOOLKIT include files
- declaring the grammar function
- setting up the SAS/TOOLKIT environment
- calling the parser and checking for errors
- calling other SAS/TOOLKIT routines
- ending a user-written procedure.

These steps are explained in detail in the remainder of this chapter.

Defining Modules

When you write a SAS procedure in PL/I, you can break your procedure into multiple source code files, as you can with any other PL/I program. At the beginning of the main module, the first statement that is not a comment should be the following:

```
%INCLUDE MACPORT6;
```

The MACPORT6 file contains many definitions needed by SAS/TOOLKIT software.
Also in the main module, include the following line to indicate that this is the main procedure source file:

```
routine: PROC PORTMAIN;
```

The line is expanded to one of the following:

```
   /* on MVS and CMS systems */
routine: PROC OPTIONS(MAIN,REENTRANT) REORDER;

   /* all other systems */
routine: PROC;
```

In all other modules that are not the main module, the first statement that is not a comment must be as follows:

```
routine: PROC(arguments) PORTPROC;
```

The line is expanded to one of the following:

```
   /* on MVS and CMS systems */
routine: PROC OPTIONS(REENTRANT) REORDER;

   /* all other systems */
routine: PROC;
```

Accessing SAS/TOOLKIT Include Files

The next step is to include the SAS/TOOLKIT include file and other necessary include files. Use these statements to access necessary definitions for writing procedures in PL/I:

```
%INCLUDE UWPROC;
%INCLUDE PROC;
```

The UWPROC include file contains many definitions, such as the special data types used in the examples in this book. The PROC file contains the procedure parsing structure, which is required in all procedures.

After these statements, you can include any other files needed for your program. In particular, if you are using any of the structures described in Chapter 20, "Data Structures," you must include the definitions of those structures. Use any of the following statements that apply:

```
%INCLUDE FATTRSTR;
%INCLUDE MEMLIST;
%INCLUDE NAMESTR;
%INCLUDE WZENV;
%INCLUDE XCOPSTR;
%INCLUDE XOOPNSTR;
```

Declaring the Grammar Function

Before you begin the main task of your program, you need to declare the grammar function as follows:

```
DCL grammar-function ENTRY RETURNS(PTR);
```

where *grammar-function* is the output of the grammar processor and is the same name as the value you specify for the GRAMFUNC= option in the PROC USERPROC statement for your grammar.

Setting Up the SAS/TOOLKIT Environment

The next executable statement in the main procedure code must be the following:

```
CALL UWPRCP(ADDR(PROC));
```

The PROC structure is defined in the PROC include file, which you have already included. The UWPRCP routine is the SAS/TOOLKIT routine that sets up the interface between your PL/I procedure and the SAS/TOOLKIT environment.

▶ ***Caution*** .... ***You must call the UWPRCP routine for your procedure to work correctly.*** ▲

Calling the Parser and Checking for Errors

Before you can access the information specified by the user in the statements that call your procedure, you must parse the statements. To parse the statements, use the SAS_XSPARSE routine and the grammar function you declared earlier.

```
CALL SAS_XSPARSE(grammar-function(),SAS_NUL(),ADDR(PROC));
```

The second argument to SAS_XSPARSE is always the null value returned by a call to SAS_NUL, and the third argument is always the address of the PROC structure.

Note: The SAS_NUL routine returns a special null value needed for SAS/TOOLKIT routines written in PL/I. Do not use the builtin NULL function of PL/I.

Immediately after you parse the statements, check for parsing errors, as in this example:

```
IF (PROC.ERROR >= XEXITPROBLEM) THEN
   CALL SAS_XEXIT(XEXITSYNTAX,0);
```

The XEXITPROBLEM and XEXITSYNTAX symbols (as well as many others) are defined in the UWPROC include file. When you exit the procedure and specify one of the XEXIT*xxxx* symbols, an appropriate error message is printed by the SAS System. Refer to the description of the SAS_XEXIT routine in Chapter 22, "SAS_X Routines," for more information on exit codes.

Calling Other SAS/TOOLKIT Routines

The main action of your program is a combination of calls to various SAS_X routines, SAS_Z routines, and standard PL/I programming statements. In general, you use SAS_X or SAS_Z routines any time you perform I/O, allocate memory, manage character strings, format data, or exit. In addition, the SAS_Z routines provide special routines for mathematical and statistical functions and for handling missing values.

Chapter 10 explains the primary routines needed in most SAS procedures. Refer to Chapter 22 for detailed reference information on the SAS_X routines and to Chapter 23, "SAS_Z Routines," for reference information on those routines.

Ending a User-Written Procedure

At any point that you want to terminate the procedure, call SAS_XEXIT. If you want to return from a non-main function, use the RETURN statement.

▶ ***Caution*** .... ***Do not use the RETURN statement to end the main function in your procedure. Never use the STOP statement.*** ▲

Call SAS_XEXIT as follows:

```
CALL SAS_XEXIT(XEXITNORMAL,0);
```

The XEXITNORMAL symbol, like the XEXITPROBLEM and XEXITSYNTAX symbols discussed earlier in this chapter, is defined in the UWPROC include file. Refer to the description of the SAS_XEXIT routine in Chapter 22 for more information on exit codes.

Summary

This section summarizes the template for a SAS/TOOLKIT procedure written in PL/I.

```
%INCLUDE MACPORT6;     /* USE IN ALL SOURCE FILES                          */

ABC: PROC PORTMAIN;    /* IN MAIN SOURCE ONLY; USE PORTPROC ELSEWHERE */
%INCLUDE UWPROC;       /* USE IN ALL SOURCE FILES                          */

%INCLUDE PROC;         /* INCLUDE IN SAME FILE AS CALL TO SAS_XSPARSE */
other %INCLUDE statements

DCL grammar-function ENTRY RETURNS(PTR); /* DECLARE GRAMMAR */
other declarations

CALL UWPRCP(ADDR(PROC));     /* FIRST STATEMENT AFTER DECLARATIONS    */

NULO = SAS_NUL();            /* OBTAIN APPROPRIATE NULL VALUE         */
CALL SAS_XSPARSE(EXMPLG(),NULO,ADDR(PROC));     /* PARSE STATEMENTS   */
IF (PROC.ERROR >= XEXITPROBLEM) THEN
   CALL SAS_XEXIT(XEXITSYNTAX,0); /* CHECK FOR SUCCESSFUL PARSING    */

CALL SAS_XEXIT(XEXITNORMAL,0);      /* NO STOP OR RETURN PERMITTED    */
END ABC;
```

Before you write the main procedure code, it's a good idea to build your grammar and the template shown above, run the USERPROC procedure, compile both the grammar and the procedure, and then test the skeletal procedure to be sure no errors occur. After you understand the grammar, basic procedure template, and the process of running PROC USERPROC, compiling, and linking, you can expand your procedure to its full functionality.

You may also want to keep this template available as a starting point for procedures you will develop later.

Chapter 9 Building a Template for SAS/TOOLKIT™ Procedures in the IBM® 370 Assembler Language

Introduction

This chapter introduces the most basic steps to writing a SAS procedure using the IBM 370 assembler language. When you finish reading this chapter, continue with Chapter 10, "Writing a SAS Procedure," for the remaining steps in building your procedure.

Setting Up a Template

To set up a template for a SAS procedure written in IBM 370 assembler, you need to include the following steps:

- communicating with SAS/TOOLKIT software
- calling the parser and checking for errors
- calling other SAS/TOOLKIT routines
- ending a user-written procedure.

These steps are explained in detail in the remainder of this chapter.

Communicating with SAS/TOOLKIT Software

The first statement in your procedure copies a special file that contains the definitions needed for SAS/TOOLKIT procedures.

```
COPY UWPROC
```

The main module begins with a CSECT of MAIN and contains the standard prologue of saving registers and establishing a save area. Following these steps, call the UWPRCC routine as follows:

```
LA     R1,PROCSTR           STRUCTURE DEFINED IN UWPROC
ST     R1,PARMLIST
LA     R1,PARMLIST
L      R15,=V(UWPRCC)
BALR   R14,R15
```

The UWPRCC routine establishes communcications between your procedure and the SAS System.

▶ ***Caution*** .... ***You must call the UWPRCC routine for your procedure to work correctly.*** ▲

Calling the Parser and Checking for Errors

Before you can access the information specified by the user in the statements that call your procedure, you must parse the statements. First, call the grammar function (the output of the USERPROC procedure), as follows:

```
SLR    R1,R1                TO ZERO PARMLIST PTR
L      R15,=V(grammar-function)
BALR   R14,R15              CALL GRAMMAR FUNCTION
```

To parse the statements, use the SAS_XSPARSE routine and the pointer returned by the grammar function in the previous step. The return value from the grammar function is stored in R15.

```
SLR    R0,R0                NULL FOR SECOND ARGUMENT
LA     R1,PROCSTR
STM    R15,R1,PARMLIST      SAVE PARMS FOR SAS_XSPARSE
LA     R1,PARMLIST
L      R15,&SAS_XSPARSE
BALR   R14,R15
```

When you call SAS_XSPARSE, R1 must point to a parameter list containing three data items:

- □ the pointer returned by the grammar function
- □ 0
- □ the address of the PROC structure (which is defined in the UWPROC file you copied into your program).

The contents of R0 are irrelevant when you make a call to SAS_XSPARSE.

Note: The call to SAS_XSPARSE and any other SAS_X or SAS_Z routine is slightly different in assembler than in other languages. You must prefix the name of the routine with an ampersand (&). This form of the call is required for the macro symbol to be expanded correctly.

Immediately after you parse the statements, check for parsing errors, as in this example:

```
         L     R0,PROCSTR+PROC_ERROR-PROC
         C     R0,=A(XEXITPROBLEM) LESS THAN XEXITPROBLEM NO ERRORS
         BL    AFTER
         SLR   R1,R1               SET REASON CODE TO 0
         STM   R15,R1,PARMLIST     SAVE VALUES IN PARMLIST FOR SAS_XEXIT
         LA    R1,PARMLIST
         L     R15,&SAS_XEXIT
         BALR  R14,R15
AFTER    DS    0H
```

The XEXITPROBLEM symbol (as well as many others) is defined in the UWPROC file. When you exit the procedure and specify one of the XEXIT*xxxx* symbols, an appropriate error message is printed by the SAS System. Refer to the description of the SAS_XEXIT routine in Chapter 22, "SAS_X Routines," for more information on exit codes.

Calling Other SAS/TOOLKIT Routines

The main action of your program is a combination of calls to various SAS_X routines, SAS_Z routines, and standard IBM 370 assembler programming statements. In general, you use SAS_X or SAS_Z routines any time you perform I/O, allocate memory, manage character strings, format data, or exit. In addition, the SAS_Z routines provide special routines for mathematical and statistical functions and for handling missing values.

Chapter 10 explains the primary routines needed in most SAS procedures. Refer to Chapter 22 for detailed reference information on the SAS_X routines and to Chapter 23, "SAS_Z Routines," for reference information on those routines.

Ending a User-Written Procedure

When you are ready to terminate the procedure, call SAS_XEXIT.

▶ ***Caution** .... **Do not use a standard epilogue to end the procedure.*** ▲

Call SAS_XEXIT as follows:

```
L     R1,XEXITNORMAL
LR    R0,R1
STM   R0,R1,PARMLIST
LA    R1,PARMLIST
L     R15,&SAS_XEXIT
BALR  R14,R15
```

The XEXITNORMAL symbol, like the XEXITPROBLEM symbol discussed earlier in this chapter, is defined in the UWPROC file. Refer to the description of the SAS_XEXIT routine in Chapter 22 for more information on exit codes.

Summary

This section summarizes the template for a SAS/TOOLKIT procedure written in IBM 370 assembler.

```
*  required in all procedures
         COPY  UWPROC
*
*  set up communications with the SAS System
         LA    R1,PROCSTR
         ST    R1,PARMLIST
         LA    R1,PARMLIST
         L     R15,=V(UWPRCC)
         BALR  R14,R15
*
*  obtain pointer from grammar function
         SLR   R1,R1
         L     R15,=V(grammar-function)
         BALR  R14,R15
*
*  parse the statements
         SLR   R0,R0
         LA    R1,PROCSTR
         STM   R15,R1,PARMLIST
         LA    R1,PARMLIST
         L     R15,&SAS_XSPARSE
         BALR  R14,R15
*
*  perform error checking after parsing
         L     R0,PROCSTR+PROC_ERROR-PROC
         C     R0,=A(XEXITPROBLEM) LESS THAN XEXITPROBLEM NO ERRORS
         BL    AFTER
         SLR   R1,R1               SET REASON CODE TO 0
         STM   R0,R1,PARMLIST      SAVE VALUES IN PARMLIST FOR SAS_XEXIT
         LA    R1,PARMLIST
         L     R15,&SAS_XEXIT
         BALR  R14,R15
AFTER    DS    0H
*
*  do not use the standard epilogue to exit
         L     R1,XEXITNORMAL
         LR    R0,R1
         STM   R0,R1,PARMLIST
         LA    R1,PARMLIST
         L     R15,&SAS_XEXIT
         BALR  R14,R15
         END
```

Before you write the main procedure code, it's a good idea to build your grammar and the template shown above, run the USERPROC procedure, assemble both the grammar and the procedure, and then test the skeletal procedure to be sure no errors occur. After you understand the grammar, basic procedure template, and the process of running PROC USERPROC, assembling, and linking, you can expand your procedure to its full functionality.

You may also want to keep this template available as a starting point for procedures you will develop later.

Chapter 10 Writing a SAS® Procedure

Introduction

This chapter presents a step-by-step discussion of the process for writing a SAS procedure. This chapter does not provide details on how to use the various SAS_X routines, SAS_Z routines, or data structures mentioned in the discussion. Refer to the detailed reference chapters in the "Reference" division of this book.

Before you attempt to use this chapter to write a procedure, you should be familiar with the process of writing a grammar and the contents of the statement structure (STMT). In addition, you need to browse the reference sections for SAS_X routines and SAS_Z routines to be aware of how they can provide more detailed information on topics that are mentioned briefly in this chapter. And finally, before reading this chapter, you need to read one of the chapters listed below to build the basic framework for the procedure in the language you have chosen:

- Chapter 6, "Building a Template for SAS/TOOLKIT Procedures in the C Language"
- Chapter 7, "Building a Template for SAS/TOOLKIT Procedures in the FORTRAN Language"
- Chapter 8, "Building a Template for SAS/TOOLKIT Procedures in the PL/I Language"
- Chapter 9, "Building a Template for SAS/TOOLKIT Procedures in the IBM 370 Assembler Language"

This chapter describes the following steps in writing a procedure:

- obtaining information from the statement structures, including data set names, options, parameters, and lists of variables specified by the user
- allocating memory
- accessing attributes of variables
- reading from and writing to SAS data sets
- BY-group processing
- creating new SAS variables
- printing output
- exiting and handling errors.

Conventions for Examples

Throughout this chapter, examples are given to illustrate the process being described. In most cases, these examples are part of a complete procedure, PROC MULTIPLY, that is described and listed at the end of the chapter. This procedure performs a very simple task; it multiplies numeric variables in a SAS data set by a value you specify and writes the new values to a new SAS data set. The MULTIPLY procedure serves as a good example for this chapter because it performs many of the tasks that require interfacing with the SAS System. For a detailed description of this procedure, refer to "Complete Example" later in this chapter.

The examples in this chapter are shown in the C language but the general steps are the same for every language. Refer to Chapter 11, "Sample Procedures in the FORTRAN, PL/I, and IBM 370 Assembler Languages," for the examples of PROC MULTIPLY in those languages.

Note: The example fragments throughout this chapter may differ slightly from the complete example at the end of the chapter. This difference occurs because the complete example performs some tasks within functions called by the main function. To simplify the explanation of example fragments, the description in the text of the chapter shows the example as if it were inline with the main function of the program.

For example, in the text some of the fragments show a value being assigned to a variable such as the following:

```
int    PRINT;     /* a boolean */

PRINT = SFOPT(proc.head, 1);
```

In the complete example, the following statement assigns the value to a dereferenced pointer:

```
int    *PRINT;     /* pointer to the boolean */

*PRINT = SFOPT(proc.head, 1);
```

Conventions for Declaring Variables

The examples in this chapter, as well as other examples throughout this book, use data types that are common to the C language. In addition, other types have been defined within SAS/TOOLKIT files. Table 10.1 shows the equivalent data type in languages other than C.

Table 10.1 *Comparable Data Types in Supported Languages*

C Data Type or Typedef ♦	FORTRAN Data Type	PL/I Data Type♦♦	IBM 370 Assembler Data Type
`long (rctype)`	INTEGER*4	FIXED BIN(31) (FXL)	F
`short (boolean)`	INTEGER*2	FIXED BIN(15) (FXS)	H
`int`♦♦♦	INTEGER*4	FIXED BIN(31)	F
`double`	REAL*8	FLOAT BIN(53) (FTL)	D
`float`	REAL*4	FLOAT BIN(21) (FTS)	E
`char[`*n*`] (char`*n*`)`	CHARACTER*n*	CHAR(*n*)	C[L*n*]
`char * (ptr)`	CADDR(*object*)	PTR	A
type `*`	IADDR(*object*)	PTR	A
`struct` *name*	COMMON	DCL 1 ...,	DSECT

♦ The data types in parentheses are defined in the UWHOST file in the standard header files for SAS/TOOLKIT software and are equivalent to the preceding standard C data type.

♦♦ The data types in parentheses are macro variable names that are defined in the MACPORT6 file in the standard PL/I macro library and are equivalent to the preceding standard PL/I data type.

♦♦♦ Many of the variables in this book are declared as type `int`. On most host operating systems, `int` is a long integer, but do not substitute `long` for `int` when you declare a variable.

Adding to the Template for Your Chosen Language

Chapters 6, 7, 8, and 9 explain how to build a template for a SAS procedure in the C, FORTRAN, PL/I, and IBM 370 assembler languages, respectively. Those chapters detail the statements needed to establish the link between your procedure, the SAS System, and the information gathered in the statement structures by the parser. This chapter continues to build on the procedure by focusing on the techniques for extracting information from the statement structures that was stored by the call to SAS_XSPARSE.

Accessing Information in the Statement Structures

When the parser completes its analysis of the input statements, it stores information about those statements in two structures: the statement structure contains the actual data from the input statements, and the parsing structure contains status codes and addresses set up by the parser. This process is described in more detail in Chapter 3, "Writing a Grammar," and is summarized briefly in the following paragraph.

In the template built in Chapters 6, 7, 8, or 9, you call SAS_XSPARSE to parse the statements entered by the user and then test the error code set by the parser. If the user invokes your procedure correctly so that the parser does not encounter any errors, you can extract the information specified by the user from the statement structures. The address of the first statement structure is stored in the `head` field of the `proc` structure. If you have created more than one statement structure in your grammar, the `next` field of the first statement structure points to the next structure. This chain of statement structures is illustrated in Figure 3.1 in Chapter 3.

Although you can have multiple statement structures for one procedure, this chapter assumes that all of the items to be extracted from the statement structures are stored in a single structure, pointed to by `proc.head`. You use this address with the SF routines to access the information stored by the parser. The next three sections show how to use the appropriate SF routine to extract information about the options, parameters, data set names, and lists of variables specified by the user. Chapter 24, "Miscellaneous Routines and Values," provides reference information for the SF routines.

Determining the Values of Arguments

In addition to the name of the statement and any lists of variables or values that are permitted by the statement, your procedure may permit a user to specify items that are typically called either arguments or options (optional arguments). When your procedure parses these arguments, it handles different types of arguments differently. In this discussion of parsing, the following terms are used to describe the different types of arguments:

bit argument
: is a SAS argument that can be turned on or off.

parameter argument
: is a SAS argument in the format *keyword=value*. These are called more simply *parameters*.

data set reference
: is a SAS argument that specifies a data set reference similar to DATA=*libref.member* or OUT=*libref.member*. Accessing this form of argument is discussed in "Opening SAS Data Sets" later in this chapter.

Note: If one of the arguments for your procedure is not optional, you should test for its presence immediately after ensuring that the parsing is successful. The grammar cannot easily ensure that one of the arguments is required; therefore, you should perform this test in your procedure's code.

Bit Arguments

To access bit arguments, use the SFOPT routine. This routine indicates that a bit argument was specified by returning a 1. If the argument was omitted, SFOPT returns a 0. Use the same number in the call to SFOPT that you used in the @OPT semantic action that defined the bit in the grammar.

To simplify your program's logic, you may want to switch the setting of the option returned by SFOPT. For example, in the sample program, the options NOPRINT and NOSUM correspond to @OPT(1) and @OPT(2) respectively in the grammar. When the NOPRINT and NOSUM options are tested, it makes the program logic simpler to set a PRINT and SUM flag, so the values returned by SFOPT are reversed to set the program flags.

```
int   PRINT,     /* a boolean */
      SUM;       /* a boolean */

PRINT = SFOPT(proc.head, 1) ? 0 : 1;
SUM = SFOPT(proc.head, 2) ? 0 : 1;
```

Parameter Arguments

To access parameter options, use one of these routines:

SFC8 — returns an eight-character string.

SFC8D — returns an eight-character string and supplies a default value when the user omits the argument.

SFF — returns a numeric parameter value.

SFFD — returns a numeric parameter value and supplies a default value when the user omits the argument.

Note: When you use one of these routines to access the value of a parameter that does not have a default value, you should always call the SFTYPE routine first to ensure that the user specified the parameter.

In the sample program, the MULT= option needs to be accessed. This option is defined with the @PARM semantic action in the grammar.

```
("MULT" "="                    @PARM(3,1)
            INT)
```

The sample program accesses the value of the parameter by using the SFFD routine, which provides a default value if none is specified. The second argument to SFFD must match the first argument to the @PARM action. The following line stores the value of the MULT= option in the program variable, **mult**:

```
double  mult;

mult = SFFD(proc.head, 3, 1.0);
```

If no value is specified by the user, the SFFD routine supplies the value of 1.0, the third argument.

Alternatively, if you do not want to provide a default value for the parameter, use the SFF routine. In this case, be sure to test the parameter to see if the user specified it before you attempt to access the value. Use the SFTYPE routine to check for a nonzero value, which indicates the user specified the parameter.

```
if (SFTYPE(proc.head, 3)) mult = SFF(proc.head, 3);
```

Opening SAS Data Sets

In most procedures, data sets are opened by the parser and closed by SAS_XEXIT. Procedures that allow the user to specify an input or output SAS data set only need to locate the information about the data sets that is stored in the statement structure. If you have included the @DSDFLT semantic action in your grammar to provide a default data set when the user omits the data set name, the SAS Supervisor automatically supplies _LAST_ for input data sets and WORK.DATA*n* for output data sets. Therefore, when you use the @DSDFLT action, your procedure does not need to test to determine if a data set name is supplied; your program can assume that a data set is available.

Before you can read from or write to a SAS data set, you must retrieve the fileid pointer for the data set. This pointer is used in all of the routines that perform data set I/O. To access pointers to data sets, use the SFFILE routine. The number used in the call to SFFILE corresponds to the first argument in the @DS or @DSDFLT semantic action coded in your grammar. In the grammar for PROC MULTIPLY, the input file and output file are coded like this:

```
"DATA" "=" DSFIELD     @DS(1,1,1,4,3,4,0)
"OUT"  "=" DSFIELD     @DS(2,2,0,2,2,4,0)
```

The grammar also defines default files for each of these SAS data sets if the user omits either the DATA= or OUT= option.

To access the fileids for the input and output SAS data sets in the procedure, use the SFFILE routine as follows:

```
ptr   infile,
      outfile;

   /* initialize the input and output files */
infile = SFFILE(proc.head, 1);
outfile = SFFILE(proc.head, 2);
```

Accessing Variable and Value Lists

Depending on how you code your grammar, some statements that the user enters include lists of variables or, occasionally, lists of values. The most common of these lists is the one specified in the VAR statement. When the parser encounters one of these lists, it stores the contents of the list in memory and then places information about the list, such as the number of listed items and the address of the list, in the statement structure.

Before your procedure can process any observations from a data set, it must determine what variables to process. The VAR statement and similar statements that allow the user to specify names of variables are stored in the statement structure as arrays of integers. These integers indicate variable numbers. Note that the variable list does *not* contain values of variables, only variable numbers. These numbers refer to the number of the variable in the *data set variable list*. The data set variable list contains the relative position of each variable in a data set. When any other list of variables is created, the variable names entered by the user are transformed into the corresponding variable numbers from the data set variable list. Your program accesses these variable numbers and uses them in calls to the SAS_XV routines to process variables in an observation. The data set variable list and other kinds of lists are described in more detail in the "Lists in the Statement Structure" section of Chapter 20, "Data Structures."

To access the variable lists, use the following routines:

SFLP
: for variable lists, returns a pointer to an array of variable numbers. If your grammar defines the list as a list of numeric values, SFLP returns a pointer to an array of numeric values.

SFN
: returns the number of elements in the array.

Example for the VAR Statement

The following example shows how the SF routines are used in the sample program to access the VAR statement. The grammar creates the VAR statement in the fourth statement structure element, so the second argument to the SFLP routine is 4. The SFLP routine returns a pointer to the array of variables specified in the VAR statement. The SFN routine determines how many variables are in that array. Both of these values are used later in the program to set up the scatter-read and gather-write operations.

```
short  *list,
       nvar;

list = (short *)SFLP(proc.head, 4);
nvar = SFN(proc.head, 4);
```

Handling BY Statement Lists

If your procedure permits the user to specify a BY statement, the grammar for the procedure includes a BYSTMT production. When you use a BYSTMT production in your grammar, a list of BY variables is automatically created as list 0 in the PROC statement's statement structure. In most cases you do not need to directly process the list of BY variables because they are handled automatically when you call the SAS_XBY routines. You may need to access the list of BY variables, however, when you create a default variable list. This topic is discussed in the following section.

Building a Default List

In many cases, your procedure must create a default list when the user does not specify one. For example, many procedures that permit the use of a VAR statement create a default list of variables when the VAR statement is not specified by the user.

Call SAS_XVDFLST to create a default list. SAS_XVDFLST only has an effect when the user has not provided a variable list. When you call SAS_XVDFLST, you can indicate if you want to exclude variables specified in other lists from the default list that you are building. For example, if you permit a BY statement, an ID statement, and a VAR statement, in most cases you don't want a default VAR list to contain the same variables that are in the BY and ID lists, because your output would include those variables twice. To avoid this duplication, you can use the fifth parameter (and subsequent ones) to specify variable lists to exclude from the default list.

In the following example, the list for the BY statement is excluded from the default variable list built for the VAR statement. Notice that before you call SAS_XVDFLST, you must first obtain the pointers to the BY and VAR list structures by calling SFLD. Then, to obtain the pointer to the variable list, call SFLP.

```
short   *list,
        nvar;
double  mult;
ptr     infile;
struct  STLIST    *bylist,
                  *varlist;
rctype  rc;

   /* create a default variable list if none is given */
bylist = (struct STLIST *)SFLD(proc.head, 0);
varlist =( struct STLIST *)SFLD(proc.head, 4);
rc = SAS_XVDFLST(infile, 1, varlist, 1, bylist);
SAS_XPRLOG(rc);

list = (short *)SFLP(proc.head, 4);
nvar = SFN(proc.head, 4);
```

Allocating Memory

When you write a SAS procedure, in most cases you cannot know in advance the number of variables that will need to be processed by the procedure. That is, a general SAS procedure, like PROC PRINT, must be able to process data sets with one to thousands of variables. To achieve this flexibility, your procedure must be able to dynamically allocate the space needed to store the variables. First, you must determine how many variables you need to store and then you must allocate the appropriate amount of space.

In addition, you may need to allocate space for arrays that vary in size each time the procedure is called. Any time your procedure needs to dynamically allocate memory, use the SAS_XM routines. To avoid fragmenting memory, it is better to use the SAS_XM routines than native language memory allocation routines.

You can use the SAS_XM routines in two ways:

- Define a pool of memory that is used each time your program needs to allocate space. You use the SAS_XMPOOLC routine to create the pool and then call SAS_XMALLOC each time you want to allocate some of the memory. The advantage of this method is that you can free all the allocated memory with a single call when you are finished using it.
- Separately allocate memory each time it is needed. Use the SAS_XMEM routines with this method. This technique is more useful if you need to allocate memory only once or twice in your program. You need to make only a single call for each allocation, but you must free memory with individual calls as well.

The example program uses calls to SAS_XMPOOLC and SAS_XMALLOC to create a pool, which is the first method described. Note that in the following example, the pool is created using default sizes for initial storage allocation and overflow allocations (XM_ISA and XM_OSA). These storage sizes vary depending on the host. In addition, the XM_EXIT argument to SAS_XMPOOLC indicates that the procedure should terminate if it cannot obtain the requested amount of memory. When you use this argument, it is not necessary to test the success of the call to SAS_XMPOOLC or the success of subsequent calls to SAS_XMALLOC.

```
ptr    pid;

   /* define a pool of memory */
pid = SAS_XMPOOLC(XM_ISA, XM_OSA, XM_EXIT);
```

Once the memory pool is allocated, the sample program uses portions of the pool by placing calls to SAS_XMALLOC. This program allocates memory for the following:

- □ an internal buffer for storing the observation, `observat`. Note that the `nvar` variable contains the number of variables in the VAR list. This value was set by a call to the SFN routine, discussed earlier in this chapter.
- □ an array of information obtained from the NAMESTR. This is discussed in more detail in the next section.
- □ working space to store variable values when printed output is requested.

These allocations are illustrated here:

```
ptr    pid;
double *observat,
       *old_obser,
       *print_sum;
struct UWINFOSTR  *varinfo;
short  nvar;
int    obssize,
       PRINT;

   /* allocate memory for observations */
obssize = sizeof(double)*nvar;
observat = (double *)SAS_XMALLOC(pid, obssize, 0);
varinfo = (struct UWINFOSTR *)SAS_XMALLOC(pid,
          (sizeof(struct UWINFOSTR))*(nvar+1), 0);

if (PRINT) {
   old_obser = (double *)SAS_XMALLOC(pid, obssize, 0);
   print_sum = (double *)SAS_XMALLOC(pid, obssize, 0);
}
```

Accessing Attributes of Variables

Programs frequently need to make processing decisions based on the attributes of a variable. For example, if your program permitted variable lists with both numeric and character data, you would need to test the variable's type before attempting to perform calculations on the variable. This particular test is not necessary for the MULTIPLY procedure, because the variable lists built for the procedure permit only numeric variables.

You also need to access the attributes of variables if you are printing the value of a variable that has a format permanently associated with it. This type of attribute information is needed in the MULTIPLY procedure.

You can determine the attributes of a variable by accessing its variable name structure (NAMESTR). When an input data set is opened, by the @DS or @DSDFLT semantic actions or by calling SAS_XOOPEN, the opening routines allocate a NAMESTR for each variable and fill in most of the fields of the NAMESTR. You then call SAS_XFFILE to obtain the remaining information that describes formats and informats for the variables. This formatting information allows you to print output using specific formats associated with each variable. A single call to SAS_XFFILE adds formatting information to all the NAMESTRs for the variables in a given list. The sample program makes the call to SAS_XFFILE only if the PRINT option is specified by the user.

```
int   PRINT;
short *list,
      nvar;

   /* load formats for printing */
if (PRINT) {
   if (SAS_XFFILE(infile, 'F', list, nvar)) {
      SAS_XPSLOG("%1zCould not load format.");
      SAS_XEXIT(XEXITERROR, 0);
   }
other steps for print files
}
```

Once the information in the NAMESTRs is complete, you can access that information when you need it.

The sample program calls SAS_XVNAME to access the variable name structures one at a time and then transfers the pertinent information for each variable into the internal array `varinfo`. The information stored in the NAMESTRs is used several times in the example program. Creating the `varinfo` array allows you to call SAS_XVNAME once to store the few data items needed by the program, and then have access to the needed data at any point in the program.

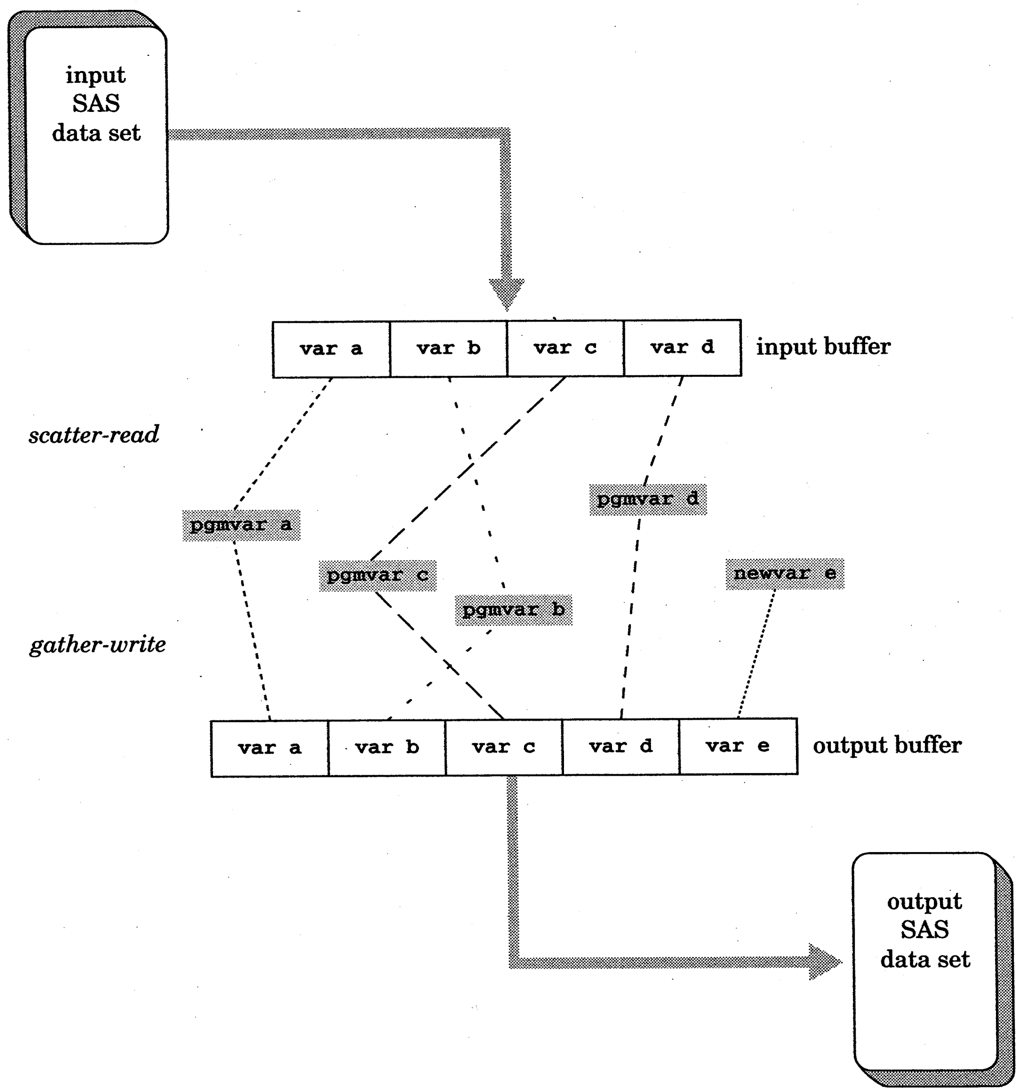
Note: The portion of the program shown below also includes a call to SAS_XVPUTD. This routine is described in detail in the next section.

```
struct NAMESTR *Nam_ptr;
ptr    infile;
short  *list,
       nvar;
double *observat;
struct SYSOPTS printopt;   /* a structure specific to this program */
struct UWINFOSTR *varinfo; /* a structure specific to this program */
int    x;


for (x = 0; x < nvar; x++) {
   SAS_XVNAME(infile, list[x], &Nam_ptr);
   SAS_XVPUTD(out_ptr, Nam_ptr, 0, observat+x, sizeof(double));
   memcpy(varinfo[x].name, Nam_ptr->nname, Nam_ptr->namelen);
   varinfo[x].namelen = Nam_ptr->namelen;
   varinfo[x].nfcode = Nam_ptr->nfcode;
}
```

Scatter-Reading and Gather-Writing Operations

When a procedure reads a SAS data set and writes a new data set, it stores the values of the variables in the program as it processes each observation. With the SAS/TOOLKIT routines, your program can easily access each variable in the observation, store the variables in noncontiguous memory, and then gather the variables again to construct an output SAS data set. The process of reading from a SAS data set is called *scatter reading* because you read a single observation and scatter its variables to various program variables. The process of writing a SAS data set is called *gather writing* because you gather the program variables to build the output observation. Figure 10.1 illustrates scatter-reading and gather-writing operations.

Figure 10.1
Scatter Reading and Gather Writing an Observation



To initialize the scatter-reading and gather-writing operations, you must call the SAS_XVGETI and SAS_XVPUTI routines, respectively. These routines return pointers that you must pass to the rest of the scatter-read and gather-write routines. The call to these routines also establishes the number of variables that should be read from or written to the data set. If you want to create new variables for the output data set, be sure that you count the number of new variables when you specify the number of output variables in the call to SAS_XVPUTI. Refer to "Creating a New SAS Variable" later in this chapter for instructions on defining new variables.

Preparing to Read from a SAS Data Set

Before your procedure can read observations into your program, you must define where each variable is stored. Use SAS_XVGETI, SAS_XVGETD, and SAS_XVGETE to set up the scatter-read operation. The example program calls

SAS_XVGETI to initialize the scatter-read process for the number of variables in the VAR list. The calls to SAS_XVGETD describe how the variables in an observation should be moved from the input buffer to the program's storage locations. In this example, each variable is stored in the `observat` array. Note that SAS_XVGETD uses the array of variable numbers, `list`, to determine which variables are in the VAR statement list.

When you have finished defining the scatter-read process, call SAS_XVGETE to signal the end of that process.

```
ptr    infile,
       ptr_xv;
short  nvar,
       *list;
int    x;
double *observat;

  /* define scatter reads    */
SAS_XVGETI(infile, nvar, &ptr_xv);
for (x = 0; x < nvar; x++)
   SAS_XVGETD(ptr_xv, list[x], 0, observat+x, sizeof(double),
              XV_NOFMT);
SAS_XVGETE(ptr_xv);
```

Your procedure does not need to check for whether the user has specified any data set options (such as DROP=, KEEP=, or RENAME=) because all this data set processing is handled by the SAS Supervisor before your program begins to work with the data set.

Note: SAS_XVGETD does not actually read observations; it simply defines which variables should be used in the operation.

Preparing to Write to a SAS Data Set

Setting up the gather-writing process is similar to defining the scatter-reading process. You must define the gather-writing process by calling SAS_XVPUTI, SAS_XVPUTD, and SAS_XVPUTE. In this example, the call to SAS_XVPUTI has one more variable than the call to SAS_XVGETI in case the SUM variable should be added to the output data set.

```
ptr    outfile,
       out_ptr;
short  nvar;

   /*   define gather writes */
SAS_XVPUTI(outfile, nvar+1, &out_ptr);
```

Before defining where to find the variables to write, call the SAS_XVNAME routine to access the information in the variable name structures. This process is described in detail in "Accessing Attributes of Variables" earlier in the chapter. Then call SAS_XVPUTD for each variable you want to write to the output data set. You do not have to read all variables in the data set, and you do not have to write all variables that were read. Using the information from the variable name structures in the call to SAS_XVPUTD ensures that the formats, labels, and other variable information from the input data set get carried over to the output data set.

```
ptr    infile,
       outfile,
       out_ptr,
       ptr_xv;
short  nvar,
       *list;
int    x;
double *observat;
struct NAMESTR *Nam_ptr;

SAS_XVPUTI(outfile, nvar+1, &out_ptr);
for(x = 0; x < nvar; x++) {
   SAS_XVNAME(infile, list[x], &Nam_ptr);
   SAS_XVPUTD(out_ptr, Nam_ptr, 0, observat+x, sizeof(double));
   copy information from NAMESTR if needed later
}
```

Note: Keep in mind that SAS_XVPUTD does not actually write observations; it simply defines which variables should be used in the operation.

Next, if the SUM option is specified by the user, create a new NAMESTR for it and then call SAS_XVPUTD to add the SUM variable to the output data set. The process of defining the new variable in a NAMESTR is described in detail in the next section, but the following example illustrates how to call SAS_XVPUTD for a new variable after the other variables have been added to the output data set:

```
ptr    infile,
       outfile,
       out_ptr;
short  nvar,
       *list;
int    x,
       SUM;
double *observat,
       sum;
struct NAMESTR *Nam_ptr,
               sumvar;

SAS_XVPUTI(outfile, nvar+1, &out_ptr);
for(x = 0; x < nvar; x++) {
   SAS_XVNAME(infile, list[x], &Nam_ptr);
   SAS_XVPUTD(out_ptr, Nam_ptr, 0, observat+x, sizeof(double));
   copy information from NAMESTR if needed later
}

if (SUM) {
   define new variable in a NAMESTR
   SAS_XVPUTD(out_ptr, &sumvar, 0, &sum, sizeof(double));
}
```

Finally, when all the variables in the output data set have been defined by calls to SAS_XVPUTD, you must call SAS_XVPUTE to signal the end of the output data set definition.

```
ptr   out_ptr;

SAS_XVPUTE(out_ptr);
```

Creating a New SAS Variable

If you want to add a new variable to an output data set, you must create a variable name structure (NAMESTR) to describe the variable. To do this, define a copy of the NAMESTR, initialize the fields of the NAMESTR by calling SAS_XVNAMEI, and then complete the name, length, and type fields of the NAMESTR, as well as any other fields you want to define for the variable. Finally, you need to call SAS_XVPUTD to include the new variable in the gather-output process.

▶ *Caution* .... ***Do not use the pointer to the NAMESTR that is returned by an earlier call to SAS_XVNAME.***
You cannot change the NAMESTRs for existing variables; you must create a new NAMESTR. ▲

In the example program, if the user wants to create the SUM variable, you use the `sumvar` copy of the NAMESTR. Initialize the NAMESTR with SAS_XVNAMEI, then move the appropriate information into the structure. In the example, you also need to store the information about the new variable in the array of information gathered from the variable name structures for later use in the program. Finally, add the new variable to the output data set by calling SAS_XVPUTD.

```
ptr    out_ptr;
short  nvar;
int    SUM;
double sum;
struct UWINFOSTR  *varinfo;

if (SUM) {
   struct NAMESTR sumvar;

      /* initialize the NAMESTR for the new variable */
   SAS_XVNAMEI(&sumvar);

      /* fill in NAMESTR information - use DOLLAR15.2 for format */
   sumvar.ntype   = 1;
   sumvar.nlng    = sizeof(double);
   sumvar.nname   = "SUM";
   sumvar.namelen = 3;
   sumvar.nlabel  = "Sum of Multiplied Values";
   sumvar.nlablen = 24;
   sumvar.nfl     = 15;
   sumvar.nfd     = 2;
```

```
   memcpy(sumvar.nform, "DOLLAR  ", 8);
   SAS_XVPUTD(out_ptr, &sumvar, 0, &sum, sizeof(double));

      /* add the SUM element to the variable information array */
   SAS_XFNAME("DOLLAR", 'F', NULL, &(varinfo[nvar].nfcode));
   memcpy(varinfo[nvar].name, sumvar.nname, 8);
   varinfo[nvar].namelen = sumvar.namelen;
}
```

In addition to the type, length, and name of the variable, this example defines the variable label with the `nlabel` and `nlablen` fields and a variable format by setting the values of `nfl`, `nfd`, and `nform`.

If you want to create more than one new variable, you can reuse the NAMESTR created for the first variable by calling SAS_XVNAMEI.

Reading from and Writing to SAS Data Sets

Reading and writing SAS data sets is quite simple after you have set up the scatter-reading and gather-writing processes. Create a loop that calls SAS_XBYGET to read the observation into the input buffer. SAS_XBYGET also tests for an end-of-file condition and stops processing when that is reached.

To transfer the observation from the input buffer to the locations specified by the scatter-read operation, call SAS_XVGET and point to the location of the input buffer. Use SAS_XVPUT to transfer the data from your procedure to the output buffer. Both of these processes are illustrated in Figure 10.1 earlier in the chapter. The following example illustrates how to read and write an observation:

```
ptr    infile,
       outfile,
       ptr_xv,
       out_ptr;
short  nvar;
int    x;

   while(SAS_XBYGET(infile) != NULL) {
      SAS_XVGET(ptr_xv, NULL);
      processing for observation, if needed
      for (x = 0; x < nvar; x++) {
         manipulate variables

         print the output
      }

         /* put the observations out to the new SAS data set */
      SAS_XVPUT(out_ptr, NULL);
      SAS_XOADD(outfile, NULL);
   }
```

Processing BY Groups

If your procedure permits BY-statement processing, enclose the SAS_XBYGET loop within a loop that calls SAS_XBYNEXT. If the user does not specify a BY statement in the statements used to call your procedure, SAS_XBYNEXT treats the entire data set like a single BY group. If the user does specify a BY statement, SAS_XBYGET returns a 0 at the end of each BY group, including the end of the data set. At the end of the BY group, your procedure drops out of the SAS_XBYGET loop, performs any special BY-group processing you include in the BY loop, and then returns to the top of the loop that calls SAS_XBYNEXT. When the procedure reaches the end of the data set, both SAS_XBYNEXT and SAS_XBYGET return 0. Since the example procedure permits BY-group processing, the SAS_XBYGET loop is within a SAS_XBYNEXT loop.

```
ptr    infile,
       outfile,
       ptr_xv,
       out_ptr;
short  nvar;
int    x;

while(SAS_XBYNEXT(infile) > 0) {
   while(SAS_XBYGET(infile) != NULL) {
      SAS_XVGET(ptr_xv, NULL);
      processing for observation, if needed
      for(x = 0; x < nvar; x++) {
         manipulate variables

         print the output
      }

         /* put the observations out to the new SAS data set */
      SAS_XVPUT(out_ptr, NULL);
      SAS_XOADD(outfile, NULL);
   }
}
```

Note: This example is similar to the one in the previous section, but it includes the SAS_XBYNEXT loop.

Manipulating Observations and Variables

Because the MULTIPLY procedure is so simple, the actual processing performed on each observation in the loop is minimal. Of particular interest, however, is the call to the SAS_ZMISS routine, which ensures that each numeric variable has a value that can be used in a calculation. If one of the variables has a missing value, the value for the SUM variable is also set to missing.

```
ptr    infile,
       outfile,
       ptr_xv,
       out_ptr;
```

```
short  nvar,
       *list;
int    x,
       allmissing,
       SUM;
double *observat,
       *old_obser,
       sum;
struct NAMESTR *Nam_ptr;

while(SAS_XBYNEXT(infile) > 0) {
   while(SAS_XBYGET(infile) != NULL) {
      SAS_XVGET(ptr_xv, NULL);
      sum=0;
      obs++;
      all_missing = 1;

         /* move a copy of the observations in observat */
         /* to old_obser */
      if (PRINT) memcpy(old_obser,observat,obssize);

      for (x=0;x<nvar;x++) {

            /* handle missing values */
         if (SAS_ZMISS(observat[x])) {
            observat[x] = MACMISSING;
         }

            /* otherwise perform calculations */
         else {
            all_missing = 0;
            observat[x] *= mult;
            if (SUM) sum += observat[x];
         }
         if (all_missing) sum = MACMISSING;
         if (PRINT) print_sum[x] = sum;
      }
      print output

         /* put the observations out to the new SAS data set */
      SAS_XVPUT(out_ptr, NULL);
      SAS_XOADD(outfile, NULL);
   }
}
SAS_XVPUTT(out_ptr);
SAS_XVGETT(ptr_xv);
```

Note: This example expands upon the ones in the last two sections to show more details of the SAS_XBYNEXT and SAS_XBYGET loops.

Note that the processing loop for each observation also includes statements to print the output, but these statements are not shown in the example. The "Printing Output" section later in this chapter discusses printing output in detail and the statements are shown in that section.

Freeing Space

Following the SAS_XBYNEXT loop, the sample program ends with calls to the SAS_XVGETT and SAS_XVPUTT routines to free the memory needed to read and write observations. It is not necessary to free this memory because it is automatically freed when you end the procedure. If you perform extensive processing after you finish reading and writing observations, and especially if you need to allocate additional memory, you may want to use the SAS_XVGETT and SAS_XVPUTT routines to ensure that your procedure does not run out of memory.

```
SAS_XVPUTT(out_ptr);
SAS_XVGETT(ptr_xv);
```

Printing Output

Printed output from SAS procedures can be routed to two output files: the SAS procedure output file and the SAS log. The SAS log is normally used to provide information about the data set being processed or to note any errors or unusual conditions. You can also print debugging messages to the log when you are testing your program. To print to the SAS log, use the SAS_XPSLOG routine to format output.

The SAS procedure output file is normally where you print report output. For procedure output, use the SAS_XP routines.

The routines you use to write procedure output depend on the language you are using to write your procedure. This chapter discusses and illustrates writing output with the C language. The section "SAS_XP Routines: Printing" in Chapter 22, "SAS_X Routines," contains a discussion of which routines are appropriate for printing output in other languages, and the examples in Chapter 11 illustrate this process.

Note: It is possible to use native language statements to print output, but for your procedure to write output to the SAS log and procedure output file, you must use the SAS_XP routines.

Using Substitution Characters to Print Output

Output messages can be written exactly as you code them or they can be formatted and edited by the SAS System. You can use substitution characters to format variable values or to change how a line is printed. The following sections describe the general method for using substitution characters. Refer to the description of the SAS_XP routines in Chapter 22 for more information on the types of substitutions permitted with the SAS_XP routines.

Formatting Variable Values

The SAS_XP routines allow you to include substitution characters in a string that is to be printed. You can then provide variable values or the results of functions to replace the substitution characters. For example, if you want to print the value of a character variable in the log, you can build the message using a substitution character to represent the value of the variable. This call to SAS_XPSLOG illustrates how to do this:

```
char[40]  custname;

SAS_XPSLOG("The value of customer name is %s.", custname);
```

Using substitution characters in the SAS_XP routines is very similar to using the C language `printf` or `sprintf` routines. You can specify any number of substitution characters in a string to be printed, but be sure that the number of variables or values that follow the string correspond to the number of substitution characters.

Field widths

In addition to simple substitutions of values, you can use substitution characters to edit or modify the string to be printed. For example, if you want to print an integer and format the value in a width of 10, you can include the field width in the substitution character, as shown here:

```
long  custnum;

SAS_XPSLOG("The value of customer number is %10ld.", custnum);
```

This example substitutes the customer number in the string and right-justifies it in a field width of 10. You might also want to specify the field width as a variable value instead of a literal number; in this case, specify an asterisk in the substitution character and then include the width as a separate argument in the call to the SAS_XP routine, as illustrated here:

```
int   width = 10;

SAS_XPSLOG("The value of customer number is %*ld.", width, custnum);
```

Note: When you use the asterisk form for a substitution character, you must specify two arguments for the substitution character: the width and the variable value.

Specifying decimal places

Printing a floating-point number with a specific field width and number of decimal places is very similar to printing integers in a specified field width. For example, to print the total purchase amount for a customer, using a field width of 12 and 2 decimal places, use the following statements:

```
double   totcost;

SAS_XPSLOG("The total purchase amount is %12.2f.", totcost);
```

You can use asterisks to represent the field width and decimal places much as you did for integers in the last section. The primary difference is that you must provide three arguments for the substitution character: the width, the number of decimal places, and the floating-point value to be formatted.

```
int      width = 12,
         dec = 2;
double   totcost;

SAS_XPSLOG("The total purchase amount is %*.*f.", width, dec, totcost);
```

Altering Lines with Substitution Characters

You can also use substitution characters to alter how lines are printed. For example, you can specify a substitution character that indicates how far to tab before printing a text string; other characters can indicate that the line should be highlighted or displayed in a special color. The example in this chapter uses the tab substitution character to position text in specific columns, as illustrated here:

```
int   obs;

SAS_XPS("%*t%d", (3+printopt.offset), obs);
```

This line prints the observation number to the procedure output file and positions it at an offset calculated within the program. The asterisk works as described earlier in "Formatting Variable Values." Thus, the value of `3+printopt.offset` is substituted for the asterisk. Assume that the value of `printopt.offset` is 0; in this case, the %t substitution character becomes %3t, so the value of `obs` is printed in the third column of the procedure output file.

Summary of Substitution Characters

Table 10.2 lists the most commonly used substitution characters. Note that there are other less common ones available. Refer to "SAS_XP Routines: Printing" in Chapter 22 for more information.

Table 10.2 *Commonly Used Substitution Characters*

Substitution Character	Other Forms	Description
%c		formats a single character. Do not use this type with a string.
%d	%*d %*n*d %−d %−*n*d	formats an integer to the default data type.
	%ld %*ld %*n*ld %−ld %−*n*ld	formats a long integer.

(*continued*)

Table 10.2 *(continued)*

Substitution Character	Other Forms	Description
%f	%*f %*w.d*f %−f %−*w.d*f	formats a floating-point number.
%*.$f		formats a floating-point number using a format code.
%n		advances to a new line.
%s	%*s	formats a string.
%*n*t	%+*n*t %−*n*t	formats text to begin in the *n* column. When you specify a minus or plus sign, the text begins *n* columns before or after the current column.
%1z		prefixes messages with ERROR:.
%2z		prefixes messages with WARNING:.
%3z		prefixes messages with NOTE:.
%%		enables you to print a single percent sign (%).

Note: In general, other forms of a formatting code use the following conventions:

%*	indicates the field width is specified as a separate argument following the value to be substituted.
%*n*	indicates the field width is specified as part of the formatting code, such as `10d` to format an integer in a field width of 10.
%−	indicates the value is right-justified.

Writing SAS Log Output

In most cases, log messages provide information to the user. They are also helpful for printing debugging messages as you are developing a procedure. In the sample program, log messages are used for two purposes: to inform the user that the procedure is not a standard SAS procedure and to print a special error message. This set of messages notifies the user that this is a user-written procedure:

```
SAS_XPSLOG("SAS/TOOLKIT SOFTWARE ** MULTIPLY PROCEDURE **.");
SAS_XPSLOG("This procedure is written in C%n");
```

This example illustrates writing a log messsage when an error occurs:

```
ptr   infile;
short *list,
      nvar;

if (SAS_XFFILE(infile, 'F', list, nvar)) {
    SAS_XPSLOG("%1zCould not load format.");
    SAS_XEXIT(XEXITERROR, 0);
}
```

The SAS_XPRLOG routine generates another very useful type of error message from the `rctype` return code values returned by some of the SAS_X routines. If the call to a SAS_X routine could fail for certain known reasons, these reasons are associated with return code values of the type `rctype`. When you call SAS_XPRLOG and pass the value of a return code of this type, SAS_XPRLOG either takes no action (if the return code is 0, for successful) or prints a descriptive message that explains the reason the call was not successful. The sample program uses the SAS_XPRLOG routine to test the success of the call that creates the default list of variables:

```
struct STLIST  bylist,
               varlist;
rctype rc;
ptr    infile;

   /* create a default variable list if none is given */
bylist = (struct STLIST *)SFLD(proc.head, 0);
varlist = (struct STLIST *)SFLD(proc.head, 4);
rc = SAS_XVDFLST(infile, 1, varlist, 1, bylist);
SAS_XPRLOG(rc);
```

Note: SAS_XPRLOG does not terminate the procedure if the return code indicates an error; it only prints the descriptive message associated with the error. To ensure that the procedure ends when an error occurs, enclose the call to SAS_XPRLOG in a block that tests the return code and calls SAS_XEXIT to exit if necessary.

Understanding Procedure Output

Each page of procedure output is divided into four areas:

- □ title lines
- □ header lines
- □ body lines
- □ footnote lines.

The following sections discuss these parts in general. "Writing Procedure Output," later in this chapter, provides detailed information on creating the body of a report.

Titles and Footnotes

Your procedure output can be affected by the TITLE and FOOTNOTE statements and by settings of some options that are specified in the OPTIONS statement. You must parse and call SAS_XPAGE before setting up any print lines to ensure that any of these statements following the PROC statement are handled correctly.

Titles and footnotes are specified by the user. They are normally printed automatically, although your procedure can suppress them using the SAS_XPOPTST routine as follows:

```
SAS_XPOPTST(XPNOFOOTS, 1);
SAS_XPOPTST(XPNOTITLE, 1);
```

A blank line always follows the titles and precedes the footnotes, if any.

If the user specifies a value for the PAGESIZE= SAS system option that is less than the sum of the number of lines needed for the title, the footnotes, and 12 lines of text for the body and the headings, then the final lines of the footnotes and titles are deleted to ensure that the body and headings are printed.

Header Lines

Header lines, such as column headings, are optionally defined by the procedure using the SAS_XPHDR routine. First, you build the strings for the headers by using SAS_XPSSTR; then you call SAS_XPHDR for each line of the header. The sample program builds a partial header line and then adds a heading for the sum column only if the SUM flag is set. When the header is completed, the call to SAS_XPHDR sets up the line as the first header line.

```
char header[256];
int  length,
     offset,
     i;

SAS_ZFILLCI(' ', header, 256);

i = SAS_XPSSTR(header,256,
              "%*tOBS %*tVARIABLE %*tPREVIOUS VALUE %*tNEW VALUE",
              (2+offset), (10+offset), (22+offset), (45+offset));

length = 54 + offset;
if (SUM) {
   SAS_XPSSTR(header+(i+13),256-(i+13), "SUM");
   length += 17;
}

SAS_XPHDR(1, header, length);
```

Headers are automatically printed at the top of each page after the title lines, followed by a blank line. (The blank line can be suppressed by using the XNOBLANK flag to the SAS_XPOPST routine.) The procedure can change the header lines at any time except between an SAS_XPAGE call and the first SAS_XPSKIP that prints something on that page.

Body Lines

You print body lines by placing text in the output buffer and calling SAS_XPPRN, SAS_XPSKIP, or SAS_XPAGE. The next section discusses how to build the body of a report in more detail.

The procedure automatically prints BY-group information if there are BY variables. There is always a blank line before and after the BY line.

Writing Procedure Output

This section discusses the tasks you need to do to print procedure output:

- centering the output on the page
- putting text into the output buffer
- formatting values
- starting a new page.

Centering Output on the Page

Most SAS procedures center printed output unless the user specifies the NOCENTER SAS system option. To center output, you must know the value of the LINESIZE= SAS system option and the width of the output you will print. In addition, you should check to ensure that the NOCENTER option is not in effect. The following function from the sample program uses the SAS_XOPT routines to check the CENTER and LINESIZE= options, as well as some other SAS system options:

```
void    Chsysopt(printopt)
struct SYSOPTS *printopt;
{
   int     rc;

   printopt->center   = SAS_XOPTBGT("CENTER", &rc);
   printopt->date     = SAS_XOPTBGT("DATE", &rc);
   printopt->label    = SAS_XOPTBGT("LABEL", &rc);
   printopt->linesize = SAS_XOPTIGT("LINESIZE", &rc);
   printopt->pagesize = SAS_XOPTIGT("PAGESIZE", &rc);
}
```

The following example uses a structure to store the resulting information. The structure can be more easily passed between functions.

```
struct  SYSOPTS {
        int     center,
                date,
                label,
                offset;
        long    linesize,
                pagesize;
} printopt;
```

The final item that needs to be added to this structure of printing information is the offset from the left margin that is needed to center the printed output. The offset from the margin is calculated by subtracting the width of the string to print from the total line size and then dividing the result in half.

To simplify the format for this report, each observation is printed in the format shown in Figure 10.2.

Figure 10.2
Format for PROC MULTIPLY Report

```
obs number   variable-1   old value   new value   running total of new values
                 .
                 .
                 .
             variable-n   old value   new value   running total of new values
```

In this case, the width of each printed line does not depend on the number of variables in the VAR statement. The width of the line changes only when the NOSUM option is specified. (By default, the SUM flag is set on. When the user specifies the NOSUM option, the SUM flag is turned off.) The following function from the sample program calculates the offset. If the NOSUM option is specified, the width of the output is 54; otherwise, the width is 72. If the NOCENTER SAS system option is specified, the function returns 0 for the offset.

```
   /*------ Find the offset for printing the print file centered ----*/
int     Fnd_offs(center, SUM, linesize)
int     center,
        SUM;
long    linesize;
{
   int     cntr,
           textcntr,
           textlen = 54,
           offset;
      /* revise length of text if the SUM variable should be printed */
   if (SUM) textlen += 18;

      /* determine the offset if the output should be centered */
   if (center) {
      cntr = linesize/2;
      textcntr = textlen/2;
      offset = cntr-textcntr;
      return(offset);
   }
   return(0);
}
```

The value returned by this function is stored in the `printopt.offset` field.

Putting Text in the Output Buffer

Once you have determined where to place the output from your procedure, you can begin formatting output. To print procedure output, you can use any of the following sets of routines:

- Call SAS_XPS to format output in the output buffer. Then call SAS_XPPRN, SAS_XPSKIP, or SAS_XPAGE to print the output. For PL/I procedures, you may want to use the SAS_XPC, SAS_XPF, and SAS_XPI routines to format output in the output buffer. These routines are discussed in more detail in Chapter 22.
- Call SAS_XPSPRN to format and print output in one step.
- Call SAS_XPSSTR to format a character string, which you can then place in the output buffer with SAS_XPS or print with SAS_XPSPRN.

The sample procedure uses SAS_XPS to format output and place it in the output buffer. The output is printed with SAS_XPSKIP and SAS_XPAGE.

The sample program prints output by building a string to contain the following information:

- the observation number (which appears only once for each observation)
- the variable name
- the variable's original value
- the new value of the variable after it has been multiplied
- a cumulative total of all new variable values in the observation.

Each of these parts of a line are formatted and placed in the output buffer with SAS_XPS. When the line is completely built, it is printed with the SAS_XPPRN routine.

For example, the following code places the observation number in the output buffer and then calls an internal function to handle the variable's name and value. If the user omitted the NOSUM option when the procedure was invoked (so the SUM flag is set), a second function is called to print the cumulative total. Each variable's line is printed by the call to SAS_XPPRN. When all variables for the observation are printed, the call to SAS_XPSKIP skips a line before the next observation is started.

```
short  nvar;
int    x,
       obs,
       PRINT,
       SUM;
double *observat,
       *old_obser,
       *print_sum;
struct UWINFOSTR  *varinfo;
struct SYSOPTS *printopt;
```

```
if (PRINT) {
   SAS_XPS("%*t%d", (3+printopt.offset), obs);
   for(x = 0; x < nvar; x++) {
      Prnt_obs(observat[x], old_obser[x], &printopt,
               &varinfo[x]);
      if (SUM) Prnt_sum(print_sum[x], printopt.offset,
                        &varinfo[nvar]);
      SAS_XPPRN();
   }
   SAS_XPSKIP(1);
}
```

Note the process used in this example to format output in a string. The example calls SAS_XPS with a string that contains various substitution characters. Substitution characters are introduced earlier in this chapter in "Using Substitution Characters to Print Output" and described in detail in the "SAS_XP Routines: Printing" section of Chapter 22.

Formatting Numeric Values

In some cases, you need to format numeric values or right-justify them or both before placing the value in the output buffer. No special formatting routines are needed to format a numeric value for printing if you want to use a simple *w.d* format. Simply specify the appropriate substitution characters in the input string to the SAS_XP routines and the substituted value is formatted. If you want to print a value using a special format, such as the DOLLAR. format, you must first obtain the format code for the format and then specify the format code in a call to SAS_XPS. This routine formats the value, justifies the value as specified by the format code, and places the value in a string that can be printed with a SAS_XP routine.

The sample program uses SAS_XPS to format the numeric value, stored in the `old_obser` array element, using the format code that was obtained by calling SAS_XFFILE and stored in the `nfcode` field of the `varinfo` array. (Earlier in this chapter, the section "Accessing Attributes of Variables" describes how to access the formats by calling SAS_XFFILE.)

Note: In this illustration, the `old_obser` and `observat` values have the type `double` not `double *` because the array element was passed to an internal function that contains a single floating-point value.

```
void    Prnt_obs(obser, old_obser, printopt, varinfo)
double  obser,
        old_obser;
struct SYSOPTS *printopt;
struct UWINFOSTR *varinfo;
{
   SAS_XPS("%*t%*s", (11+printopt->offset), varinfo->namelen,
           varinfo->name);
   SAS_XPS("%*t%14.$f", (22+printopt->offset), varinfo->nfcode,
           old_obser);
   SAS_XPS("%*t%14.$f", (40+printopt->offset), varinfo->nfcode,
           obser);
}
```

In this example, the offset needed to center output on the page is stored in `printopt.offset`. The first argument after the string is substituted for the asterisk in the %t character to indicate which column the text should be printed in. The second argument, `varinfo.nfcode`, is substituted for the dollar sign in the %f character to indicate the format code for printing the value. Note that when you are substituting a format code, not a width or number of decimal places, you must use a dollar sign, not an asterisk in the substitution character.* The final argument is the value to be formatted and printed in a field width of 14.

Figure 10.3 illustrates the substitution process.

Figure 10.3
Substituted Values

```
% * t % 14. $ f
  -     --- - -
  ^      ^  ^ ^
  |      |  | |_____ floating-point value
  |      |  |
  |      |  |_______ format code (varinfo.nfcode)
  |      |
  |      |__________ field width
  |
  |_________________ column position for the tab (22+printopt.offset)
```

Starting a New Page

You can start new pages either explicitly or by default when one page of output is full. Any time you print output to the first line on a new page, the SAS System performs these actions before the line is printed:

□ If necessary, it opens the print file; if the file is already open, it issues a page feed.

□ It prints the titles, if any, and skips a line.

□ It prints the headers, if you have defined any with the SAS_XPHDR routine and skips a line (unless you have overridden that action with the SAS_XPOPTST routine).

□ It sets the number of body lines left to print on a page. The number of body lines is determined by the PAGESIZE= SAS system option. You can test this value by calling SAS_XPOPTGT with the XPLEFT mnemonic, as illustrated in the next paragraph.

If you are not concerned about creating logical pages of data, you can let the SAS System calculate when page breaks are needed. In this case, simply call SAS_XPSKIP to move to the next line of output. If you call SAS_XPSKIP and there are no lines left on the page, a new page is automatically started and the actions described earlier in this section are performed. SAS_XPPRN works in the same manner as SAS_XPSKIP except that you cannot skip multiple lines with SAS_XPPRN.

You may want to start a new page for each BY group, or you may want to determine how many lines are left on a page and start a new page if you cannot print a logical group of data. In either case, call SAS_XPAGE to skip to a new

* The dollar sign indicates that a format code, which must be of type `long`, will be substituted. The asterisk is used to substitute values of type `int`, which may be equivalent to a `short` on some systems. You must use the dollar sign to ensure that a long integer is used.

page. You can determine the number of lines left to print on the current page by calling SAS_XPOPTGT as follows:

```
int lines;

lines = SAS_XPOPTGT(XPLEFT, NULL);
```

SAS_XPAGE avoids blank or empty (just titles and footnotes) pages, even if you call it twice in a row. SAS_XPAGE does not cause a page feed, but the next attempt to actually print something triggers the page feed and prints titles and headings. SAS_XPAGE checks the printing options and updates line size, when necessary.

You should always call SAS_XPAGE after parsing but before formatting your output. This ensures that an OPTIONS statement following the PROC statement that affects printed output is handled properly.

Exiting Normally or with Errors

Throughout your procedure, you may want to test certain conditions for possible errors. If you detect an error in the input data or some other problem that is not detected by the parser, you can exit from the procedure by calling SAS_XEXIT and pass a return code indicating the cause of the problem. Refer to the description of SAS_XEXIT in Chapter 22 for a list of the valid return codes.

You should always check for errors returned by the call to SAS_XSPARSE, as shown here:

```
ptr multg  U_PARMS((void));

SAS_XSPARSE(multg(), NULL, &proc);
if (proc.error >= XEXITPROBLEM)
   SAS_XEXIT(XEXITSYNTAX, 0);
```

You should also check for errors if you allocate memory without using the XM_EXIT flag in the call to the SAS_XM routine. The sample program uses the XM_EXIT flag in the call to SAS_XMPOOLC. However, if you omit this flag, the following statements ensure that the call to SAS_XMPOOLC successfully allocated a pool of memory to be used by the procedure. The XEXITMEMORY mnemonic is used with SAS_XEXIT so that an appropriate message is printed when the allocation fails.

```
ptr     tmp = NULL;

tmp = SAS_XMPOOLC(XM_ISA, XM_OSA, 0);
if (tmp == NULL)
   SAS_XEXIT(XEXITMEMORY, 0);
```

In both of these cases, you might want to generate messages describing the error in more detail. Use SAS_XPSLOG to write messages explaining errors.

You can also generate useful information about errors by using the SAS_XPRLOG routine with any routine that returns a value of type `rctype`. All of the values for `rctype` return codes have special meanings. SAS_XPRLOG translates the return code into one or more meaningful messages about the error that occurred. The SAS_XC, SAS_XD, SAS_XO, and SAS_XV routines return `rctype` values. When you pass the value returned by one of these routines to SAS_XPRLOG, it prints a message if the value is not 0 (meaning the call that returned the `rctype` value was unsuccessful) and does nothing if the returned value is 0 (successful).

You also use SAS_XEXIT to exit normally from a procedure. If the data set is empty, you should exit from the procedure, but do not treat this as an error. The user may be testing a stream of SAS statements with the OBS=0 and NOREPLACE SAS system options.

```
int   hasobs;
long  numobs;
ptr   infile,
      outfile;

infile = SFFILE(proc.head, 1);
outfile = SFFILE(proc.head, 2);
SAS_XOINFO(infile, XO_ANY, (ptr)&hasobs);
if (hasobs == 0) {
   SAS_XPSLOG("The input data set is empty.");
   SAS_XEXIT(XEXITNORMAL, 0);
}
```

When you exit from your procedure, normally or with errors, SAS_XEXIT closes all files, empties the contents of the output buffer, and prints the current log and listing pages.

▶ *Caution* .... ***Never use the native language stop statement in a SAS procedure. In addition, do not use a return statement in the main function of a SAS procedure.*** ▲

Summary

For a procedure that reads a SAS data set and writes a new one, you usually take the following steps:

- □ Obtain the parameters and options from the statement structure.
- □ Obtain the pointers to the data sets from the statement structure.
- □ Call SAS_XVLIST or SAS_XVDFLST to set up variable lists.
- □ Allocate space for any necessary arrays by using the SAS_XM routines.
- □ Call SAS_XVNAME and SAS_XFFILE to place variable names and other information in the variable name structure.
- □ Use SAS_XVGETI, SAS_XVGETD, and SAS_XVGETE to specify which variables will be read from the input data set and where their values will be stored in the program.
- □ Define any new variables created by your procedure.
- □ Specify the variables in the output data set using SAS_XVPUTI, SAS_XVPUTD, and SAS_XVPUTE.

- □ Use a DO loop calling SAS_XBYNEXT for BY processing.
- □ Call SAS_XBYGET to read each observation into the buffer.
- □ Call SAS_XVGET to transfer values of the variables from the buffer to storage allocated by the procedure.
- □ Perform the main actions of your procedure.
- □ Call SAS_XVPUT to transfer values from the procedure to the output data set.
- □ Call the SAS_XP and SAS_XPSLOG routines to print output to the SAS listing and log files.
- □ Call SAS_XEXIT to exit from the procedure.

Complete Example

To draw together all of the elements of procedure writing, this section provides

- □ a description of the procedure in standard SAS documentation format
- □ a grammar module
- □ a complete procedure listing
- □ test statements and procedure output.

Description of the MULTIPLY Procedure

This section describes the procedure illustrated in this chapter. Because the VAR statement and BY statement are familiar to SAS users, the documentation does not describe these statements in great detail.

Abstract

The MULTIPLY procedure multiplies all or selected numeric variables from a SAS data set by a specified value and optionally sums the results. An output data set is created and the values are optionally printed.

Specifications

The following statements control the MULTIPLY procedure:

PROC MULTIPLY *options*;
  VAR *variables*;
  BY *variables*;

PROC MULTIPLY statement

PROC MULTIPLY *options*;

The following options can appear in the PROC MULTIPLY statement:

DATA=*SAS-data-set*
: names the SAS data set to be printed. If DATA= is omitted, the most recently created SAS data set is used.

MULTIPLY=*n*
MULT=*n*
: tells PROC MULTIPLY to multiply all numeric variables by this value (*n*). The value specified can be an integer or a real number.

NOPRINT
NOP
: tells PROC MULTIPLY not to print any output. Use NOPRINT when PROC MULTIPLY is to create only an output data set.

NOSUM
: tells PROC MULTIPLY not to sum the results of the multiplication.

OUT=*SAS-data-set*
: names the output SAS data set. If OUT= is omitted, PROC MULTIPLY names the output data set according to the DATA*n* convention.

VAR statement

VAR *variables*;

The VAR statement lists the variables to be used to create the new data set. Any variable in the data set can be specified, but only the numeric variables are printed since they are the only ones affected by the multiplication. If the VAR statement is not specified, all numeric variables in the data set are used.

BY statement

BY *variables*;

A BY statement can be used with PROC MULTIPLY to obtain separate printed pages for groups of observations defined by the BY variables. When a BY statement appears, the procedure expects the input data set to be sorted in the order of the BY variables. If your input data set is not sorted in ascending order, use the SORT procedure with a similar BY statement to sort the data, or, if appropriate, use the BY statement options NOTSORTED or DESCENDING.

Grammar Statements

The grammar listing shown here can be used with the MULTIPLY procedure in any language. When you run the USERPROC procedure, you simply specify the language used to write the main procedure and the grammar function is created in the same language.

Listing 10.1 *Grammar Statements for the MULTIPLY Procedure*

```
#--------------GRAMMER FOR multiply.C ---------------------#
%INCLUDE STUBGRM.

PROGRAM     =   ANYSTMT ENDJB,

ANYSTMT     =   SMULTSTMT
            |   VARSTMT
            |   BYSTMT
            ,

# STATEMENT DEF #

SMULTSTMT   =                               @PROCINIT
                                            @STMTINIT(4)
                "MULTIPLY"
                SMULTOPT*                   @DSDFLT (1,1,1,4,3,4,0)
                                            @DSDFLT (2,2,0,2,2,4,0)
                                            @STMTEND
            ,

SMULTOPT    =   ("DATA" "=" DSFIELD         @DS (1,1,1,4,3,4,0))
            |   ("OUT"  "=" DSFIELD         @DS (2,2,0,2,2,4,0))
            |   ("MULT" "="                 @PARM(3,1)
                              INT)
            |   (("NOPRINT" | "NOP")        @OPT(1))
            |   ("NOSUM"                    @OPT(2))
            ,

VARSTMT     =                               @STMTPROC
                                            @STMTLIST(4,1)
                ("VAR" | "VARIABLES")
                VARLIST
            .
```

Procedure Listing

The complete procedure listing in C is shown next. Note that when values are set in an internal function, instead of within the main function of the procedure, the form of statements such as the following will change:

```
short nvar;

nvar = SFN(proc.head, 4);
```

The statements change to the following:

```
short *nvar;

*nvar = SFN(proc.head, 4);
```

This change is necessary when the function needs to set the value of multiple variables. The address of the variable is passed to the function. It dereferences the address, stores the new value there, and the main function can then dereference the addess to get the updated value.

Listing 10.2
C Version of the MULTIPLY Procedure

```
#define MAINPROC     1
#define SASPROC      1

#include <uwproc>

ptr multg  U_PARMS((void));

void  Init_var  U_PARMS((short **, short *, double *, ptr ));
void  Out_head  U_PARMS((int, int));
void  Prt_env  U_PARMS((int *, int *, struct SYSOPTS *, ptr ));
void  Chsysopt  U_PARMS((struct SYSOPTS * ));
void  Prnt_obs  U_PARMS((double, double, struct SYSOPTS *,
                        struct UWINFOSTR *));
void  Prnt_sum  U_PARMS((double, int, struct UWINFOSTR *));
int   Fnd_offs  U_PARMS((int, int, long ));

struct  SYSOPTS {
        int     center,
                date,
                label,
                offset;
        long    linesize,
                pagesize;
};

struct  UWINFOSTR {
        int     namelen;
        char    name[8];
        long    nfcode;
};

void    U_MAIN(multiply)()
{
   ptr      infile,      /* pointer to the input data set.      */
            outfile,     /* pointer to the output data set.     */
            ptr_xv,      /* pointer for SAS_XVGET calls         */
            out_ptr,     /* pointer for SAS_XVPUT calls         */
            pid;         /* pointer to the pool of memory where */
                         /* all the allocated memory comes from.*/

   short    *list,       /* the list of variable numbers.       */
            nvar;        /* the number of variables in the list.*/

   int      x,           /* a counter variable.                 */
            obs,         /* a counter of the # of observations. */
            SUM,         /* a boolean                           */
                         /*     1= print sum in printed output  */
                         /*     0= do not print sum in output.  */
            PRINT,       /* a boolean                           */
                         /*     1= output to a print file       */
                         /*     0= Do not output to print file.*/
   all_missing,          /* a boolean                           */
                         /*  1= all the variables were missing  */
                         /*  an observation.                    */
                         /*  0= there was at least one obs      */
                         /*     that was not missing.           */
            obssize;     /* hold the size of all the            */
                         /* observations for allocating memory  */
                         /* and block moving the observations   */


   double   mult,        /* the number to multiply by.          */
            *observat,   /* pointer to the memory that will     */
                         /* hold the observations.              */
            *old_obser,  /* hold the old observations after     */
                         /* multiplying                         */
            sum,         /* the sum of all the variables for one*/
                         /* observation.                        */
            *print_sum;  /* hold the running sum for the        */
                         /* variables in the observation        */

   struct SYSOPTS printopt;

   struct UWINFOSTR *varinfo;

   struct NAMESTR *Nam_ptr; /* a pointer for retrieving the NAMESTR */
                            /* information for variables in the     */
                            /* input data set                       */

   UWPRCC(&proc);

      /* Parse the PROC Statements */
   SAS_XSPARSE(multg(), NULL, &proc);
   if (proc.error >= XEXITPROBLEM)
       SAS_XEXIT(XEXITSYNTAX, 0);


   SAS_XPSLOG("SAS/TOOLKIT SOFTWARE ** MULTIPLY PROCEDURE **.");
   SAS_XPSLOG("This procedure is written in C%n");
```

```
   /* This call to SAS_XPAGE  is necessary to check any OPTIONS */
   /* statements within the PROC call.   */
SAS_XPAGE();

   /* Initialize the input and output files */
infile = SFFILE(proc.head, 1);
outfile = SFFILE(proc.head, 2);

   /* gather information from statement structures and set up the */
   /* printing environment based on the options                   */
Init_var(&list, &nvar, &mult, infile);
Prt_env(&SUM, &PRINT, &printopt, infile, list, nvar);

   /* Define a pool of memory */
pid = SAS_XMPOOLC(XM_ISA, XM_OSA, XM_EXIT);

   /* allocate memory for observations */
obssize = sizeof(double)*nvar;
observat = (double *)SAS_XMALLOC(pid, obssize, 0);
varinfo = (struct UWINFOSTR *)SAS_XMALLOC(pid,
          (sizeof(struct UWINFOSTR))*(nvar+1), 0);

if (PRINT) {
   old_obser = (double *)SAS_XMALLOC(pid, obssize, 0);
   print_sum = (double *)SAS_XMALLOC(pid, obssize, 0);
}

   /* define scatter reads */
SAS_XVGETI(infile, nvar, &ptr_xv);
for(x=0; x<nvar; x++)
   SAS_XVGETD(ptr_xv, list[x], 0, observat+x, sizeof(double),
              XV_NOFMT);
SAS_XVGETE(ptr_xv);

   /* define gather writes and store info from the NAMESTR  */
SAS_XVPUTI(outfile, nvar+1, &out_ptr);
for(x=0; x<nvar; x++) {
   SAS_XVNAME(infile, list[x], &Nam_ptr);
   SAS_XVPUTD(out_ptr, Nam_ptr, 0, observat+x, sizeof(double));
   memcpy(varinfo[x].name, Nam_ptr->nname, Nam_ptr->namelen);
   varinfo[x].namelen = Nam_ptr->namelen;
   varinfo[x].nfcode = Nam_ptr->nfcode;
}

   /* create new variable to hold the sum */
if (SUM) {
   struct NAMESTR sumvar;

      /* initialize the NAMESTR for the new variable */
   SAS_XVNAMEI(&sumvar);

      /* fill in NAMESTR information - use DOLLAR15.2 for format */
   sumvar.ntype = 1;
   sumvar.nlng  = sizeof(double);
   sumvar.nname="SUM";
   sumvar.namelen  = 3;
   sumvar.nlabel="Sum of multiplied Values";
   sumvar.nlablen  = 24;
   sumvar.nfl  = 15;
   sumvar.nfd  = 2;
   memcpy(sumvar.nform, "DOLLAR  ", 8);
   SAS_XVPUTD(out_ptr, &sumvar, 0, &sum, sizeof(double));

     /* add the SUM element to the variable information array */
   SAS_XFNAME("DOLLAR", 'F', NULL, &(varinfo[nvar].nfcode));
   memcpy(varinfo[nvar].name, sumvar.nname, 8);
   varinfo[nvar].namelen = sumvar.namelen;
}

   /* complete gather write process */
SAS_XVPUTE(out_ptr);

obs=0;


   /* read and write observations */
while(SAS_XBYNEXT(infile) > 0) {
   while(SAS_XBYGET(infile) != NULL) {
      SAS_XVGET(ptr_xv, NULL);
      sum=0;
      obs++;
      all_missing = 1;

         /* move a copy of the observations in observat */
         /* to old_obser */
      if (PRINT) memcpy(old_obser,observat,obssize);
```

(continued on next page)

Listing 10.2 (continued)

```
        for(x=0;x<nvar;x++) {

              /* handle missing values */
           if (SAS_ZMISS(observat[x])) {
               observat[x] = MACMISSING;
           }

              /* otherwise perform calculations */
           else {
             all_missing = 0;
             observat[x] *= mult;
             if (SUM) sum += observat[x];
           }
           if (all_missing) sum = MACMISSING;
           if (PRINT) print_sum[x] = sum;
        }

           /* print the observations to the print file */
        if (PRINT) {
           SAS_XPS("%*t%d", (3+printopt.offset), obs);
           for(x=0;x<nvar;x++) {
               Prnt_obs(observat[x], old_obser[x], &printopt,
                        &(varinfo[x]));
               if (SUM) Prnt_sum(print_sum[x], printopt.offset,
                                 &(varinfo[nvar]));
               SAS_XPPRN();
           }
           SAS_XPSKIP(1);
        }

           /* put the observations out to the new SAS data set */
        SAS_XVPUT(out_ptr, NULL);
        SAS_XOADD(outfile, NULL);
      }
   }
   SAS_XVPUTT(out_ptr);
   SAS_XVGETT(ptr_xv);
   SAS_XEXIT(XEXITNORMAL, 0); /* No return or stop */
}


   /*--Access the variable lists----------------------------------*/
void    Init_var(list, nvar, mult, infile)
short   **list,
        *nvar;
double  *mult;
ptr     infile;
{
   struct  STLIST    *bylist,
                     *varlist;
   rctype  rc;

      /* Creating default variable list if none is given. */
   bylist =(struct STLIST *)SFLD(proc.head, 0);
   varlist =(struct STLIST *)SFLD(proc.head, 4);
   rc = SAS_XVDFLST(infile, 1, varlist, 1, bylist);
   SAS_XPRLOG(rc);

      /* obtain MULT= option and list of variables */
   *mult = SFFD(proc.head, 3, 1.0);
   *list = (short *)SFLP(proc.head, 4);
   *nvar = SFN(proc.head, 4);
}

   /*------ Establish printing environment -----------------*/
void    Prt_env(SUM, PRINT, printopt, infile, list, nvar)
int     *SUM,
        *PRINT;
struct SYSOPTS *printopt;
ptr     infile;
{
   *PRINT = SFOPT(proc.head, 1) ? 0 : 1;
   *SUM = SFOPT(proc.head, 2) ? 0 : 1;

   if (*PRINT)
      Chsysopt(printopt);

      /* Load formats for printing */
   if (*PRINT) {
      if (SAS_XFFILE(infile, 'F', list, nvar)) {
         SAS_XPSLOG("%1zCould not load format.");
         SAS_XEXIT(XEXITERROR, 0);
      }

        /* find indention for centering */
      printopt->offset = Fnd_offs(printopt->center, *SUM,
                         printopt->linesize);

      Out_head(*SUM, printopt->offset);
   }

}
```

```
   /*------ Check system Options ---------------------------------*/
void    Chsysopt(printopt)
struct SYSOPTS *printopt;
{
   int     rc;

   printopt->center   = SAS_XOPTBGT("CENTER", &rc);
   printopt->date     = SAS_XOPTBGT("DATE", &rc);
   printopt->label    = SAS_XOPTBGT("LABEL", &rc);
   printopt->linesize = SAS_XOPTIGT("LINESIZE", &rc);
   printopt->pagesize = SAS_XOPTIGT("PAGESIZE", &rc);
}

   /*------ Find the offset for printing the print file centered -*/
int     Fnd_offs(center, SUM, linesize)
int     center,
        SUM;
long    linesize;
{
   int     cntr,
           textcntr,
           textlen = 54,
           offset;

      /* revise length of text if the SUM variable should be printed */
   if (SUM) textlen +=18;

      /* determine the offset if the output should be centered */
   if (center) {
      cntr = linesize/2;
      textcntr = textlen/2;
      offset = cntr - textcntr;
      return(offset);
   }
   return(0);
}

   /*------ Create the headings for the variables ----------------*/
void    Out_head(SUM, offset)
int     SUM,
        offset;
{
   char header[256];
   int  length,
        i;

   SAS_ZFILLCI(' ', header, 256);

   i = SAS_XPSSTR(header,256,
                  "%*tOBS %*tVARIABLE %*tPREVIOUS VALUE %*tNEW VALUE",
                  (2+offset), (10+offset), (22+offset), (45+offset));

   length = 54 + offset;
   if (SUM) {
      SAS_XPSSTR(header+(i+13),256-(i+13), "SUM");
      length += 17;
   }

   SAS_XPHDR(1, header, length);
   SAS_XPAGE();
}

   /*----Format the numbers and print them to the print file ------*/
void    Prnt_obs(obser, old_obser, printopt, varinfo)
double  obser,
        old_obser;
struct SYSOPTS *printopt;
struct UWINFOSTR *varinfo;
{
   SAS_XPS("%*t%*s", (11+printopt->offset), varinfo->namelen,
           varinfo->name);
   SAS_XPS("%*t%14.$f", (22+printopt->offset), varinfo->nfcode,
           old_obser);
   SAS_XPS("%*t%14.$f", (40+printopt->offset), varinfo->nfcode,
           obser);
}

   /*--------- Add the sum to the output if necessary -------------*/
void    Prnt_sum(sum, offset, varinfo)
double  sum;
int     offset;
struct UWINFOSTR *varinfo;
{
   SAS_XPS("%*t%15.$f", (55+offset), varinfo->nfcode+2, sum);
}
```

Output from the MULTIPLY Procedure

The following statements were executed to illustrate PROC MULTIPLY:

```
options linesize=76 nodate;
data sales;
   input year $4. dept $ qtr1 qtr2 qtr3 qtr4;
   format qtr1 qtr2 qtr3 qtr4 dollar15.2;
   cards;
1990 parts 546.54 5854.25 89832.44 5846.01
1990 parts 4565.00 987.36 486.00 400.00
1990 parts 56.00 80.25 90.00 100.00
1990 repairs 5486.32 87800.36 98500.36 4000.01
1990 repairs 5486.36 4578.25 456.98 712.84
1990 tools 5486.23 8799.36 7100.00 8000.00
1990 tools 458.36 898.00 400.10 5000.00
;

title "Test the MULTIPLY Procedure";
proc multiply mult=5 data=sales;
   var qtr1 qtr2 qtr3 qtr4;
run;
```

The output produced by PROC MULTIPLY is illustrated in Output 10.1.

Output 10.1 *Sample Output from the MULTIPLY Procedure*

```
                         Test the MULTIPLY Procedure                      1

  OBS      VARIABLE    PREVIOUS VALUE           NEW VALUE            SUM

   1       QTR1               $546.54           $2,732.70      $2,732.70
           QTR2             $5,854.25          $29,271.25     $32,003.95
           QTR3            $89,832.44         $449,162.20    $481,166.15
           QTR4             $5,846.01          $29,230.05    $510,396.20

   2       QTR1             $4,565.00          $22,825.00     $22,825.00
           QTR2               $987.36           $4,936.80     $27,761.80
           QTR3               $486.00           $2,430.00     $30,191.80
           QTR4               $400.00           $2,000.00     $32,191.80

   3       QTR1                $56.00             $280.00        $280.00
           QTR2                $80.25             $401.25        $681.25
           QTR3                $90.00             $450.00      $1,131.25
           QTR4               $100.00             $500.00      $1,631.25

   4       QTR1             $5,486.32          $27,431.60     $27,431.60
           QTR2            $87,800.36         $439,001.80    $466,433.40
           QTR3            $98,500.36         $492,501.80    $958,935.20
           QTR4             $4,000.01          $20,000.05    $978,935.25

   5       QTR1             $5,486.36          $27,431.80     $27,431.80
           QTR2             $4,578.25          $22,891.25     $50,323.05
           QTR3               $456.98           $2,284.90     $52,607.95
           QTR4               $712.84           $3,564.20     $56,172.15

   6       QTR1             $5,486.23          $27,431.15     $27,431.15
           QTR2             $8,799.36          $43,996.80     $71,427.95
           QTR3             $7,100.00          $35,500.00    $106,927.95
           QTR4             $8,000.00          $40,000.00    $146,927.95

   7       QTR1               $458.36           $2,291.80      $2,291.80
           QTR2               $898.00           $4,490.00      $6,781.80
           QTR3               $400.10           $2,000.50      $8,782.30
           QTR4             $5,000.00          $25,000.00     $33,782.30
```

Chapter 11 Sample Procedures in the FORTRAN, PL/I, and IBM® 370 Assembler Languages

Introduction

Chapter 10, "Writing a SAS Procedure," describes how to write a simple SAS procedure by illustrating the MULTIPLY procedure written in the C language. This chapter illustrates the MULTIPLY procedure in the FORTRAN, PL/I, and IBM 370 assembler languages. The sample programs in these languages follow the same format as the C language version. You can match the descriptions of the tasks in Chapter 10 with the sample programs in this chapter.

The MULTIPLY Procedure in the FORTRAN Language

The sample listing of the MULTIPLY procedure in FORTRAN follows.

Listing 11.1
FORTRAN Version of the MULTIPLY Procedure

```
VMS  SUBROUTINE MULTIPLY
     INCLUDE (PROTOSF)
     INCLUDE (PROC)
     INCLUDE (NAMESTR)
     INTEGER*4 MULTG
     INTEGER*4 VARSIZE
     COMMON/SIZES/VARSIZE
     LOGICAL*1 CENTER
     INTEGER*4 DATE,LABEL,OFFSET,LINESIZE,PAGESIZE
     COMMON/PRINTOP/DATE,LABEL,OFFSET,LINESIZE,PAGESIZE,CENTER
     INTEGER*4 VNAMELEN,VNAMEPTR,VNFCODE
     COMMON/VARINFO/VNAMELEN,VNAMEPTR,VNFCODE
     INTEGER*4 INFILE,OUTFILE,PTRXV,OUTPTR,PID
     INTEGER*2 JS
     INTEGER*4 NVAR
     INTEGER*4 NULO
     INTEGER*4 X,I,J,OBS,IOBSER,IOLD
     LOGICAL*1 SUMOPT,PRINT,ALLMISS
     REAL*8    MULT,SUM,TEMP
     INTEGER*4 LISTPTR,OBSERVATPTR,OLDOBSERPTR,
    +          PRINTSUMPTR,VARINFOPTR,PRINTOPTPTR,VPTR
     INTEGER*4 OBSSIZE
     REAL*8    MISSVAL
     REAL*8    RBASE(1)
     CHARACTER*8 SUMNAME
```

(continued on next page)

Listing 11.1 (continued)

```
      CHARACTER*40 SUMNLABEL
      DATA SUMNAME/'SUM'/,SUMNLABEL/'SUM OF MULTIPLIED VALUES'/
      INTEGER*4 NAMPTR
C
      CALL UWPRCF(IADDR(PROCHEAD))
      NULO = 0
C
C     Parse the PROC Statements
      RC = SAS_XSPARSE(MULTG(0),NULO,IADDR(PROCHEAD))
      IF (PROCERROR .GE. XEXITPROBLEM)
     +   CALL SAS_XEXIT(XEXITSYNTAX, 0)
C
      CALL SAS_XPSLOG('SAS/TOOLKIT SOFTWARE ** MULTIPLY PROCEDURE **.')
      CALL SAS_XPSLOG('This procedure is written in FORTRAN.%n')
C
C      This call to SAS_XPAGE    is necessary to check any OPTIONS
C      statements within the PROC call.
C
      CALL SAS_XPAGE
C
C      Initialize the input and output files
      INFILE = SFFILE(PROCHEAD, 1)
      OUTFILE = SFFILE(PROCHEAD, 2)
C
C      Define a pool of memory
      PID = SAS_XMPOOLC(XM_ISA, XM_OSA, XM_EXIT)
      I = IADDR(PAGESIZE) - IADDR(CENTER) + 4
      PRINTOPTPTR = SAS_XMALLOC(PID,I,0)
C
C      gather information from statement structures and set up the
C      printing environment based on the options
      CALL INITVAR(LISTPTR, NVAR, MULT, INFILE)
C
      CALL PRTENV(SUMOPT, PRINT, PRINTOPTPTR, INFILE, LISTPTR, NVAR)
C
C      allocate memory for observations
      OBSSIZE = 8 * NVAR
      OBSERVATPTR = SAS_XMALLOC(PID, OBSSIZE, 0)
      VARSIZE = IADDR(VNFCODE) - IADDR(VNAMELEN) + 4
      VARINFOPTR = SAS_XMALLOC(PID,VARSIZE*(NVAR+1),0)
C
      IF (.NOT. PRINT) GOTO 100
         OLDOBSERPTR = SAS_XMALLOC(PID,OBSSIZE,0)
         PRINTSUMPTR = SAS_XMALLOC(PID,OBSSIZE,0)
 100     CONTINUE
C
C       define scatter reads
      RC = SAS_XVGETI(INFILE, NVAR, IADDR(PTRXV))
      DO 200 X=1,NVAR
         CALL SAS_ZMOVEI(LISTPTR+(X-1)*2,IADDR(JS),2)
         J = JS
         RC = SAS_XVGETD(PTRXV,J,0,OBSERVATPTR+(X-1)*8,
     +                   8,XV_NOFMT)
 200     CONTINUE
      RC = SAS_XVGETE(PTRXV)
C
C       define gather writes and store info from the NAMESTR
      RC = SAS_XVPUTI(OUTFILE, NVAR+1,IADDR(OUTPTR))
      VPTR = VARINFOPTR
      DO 300 X=1,NVAR
         CALL SAS_ZMOVEI(LISTPTR+(X-1)*2,IADDR(JS),2)
         J = JS
         RC = SAS_XVNAME(INFILE,J,IADDR(NAMPTR))
         CALL SAS_ZMOVEI(NAMPTR,IADDR(NTYPE),XV_NAMEL)
         RC = SAS_XVPUTD(OUTPTR,NAMPTR,0,OBSERVATPTR+(X-1)*8,
     +                   8)
         VNAMEPTR = NNAME
         VNAMELEN = NAMELEN
         VNFCODE  = NFCODE
         CALL SAS_ZMOVEI(IADDR(VNAMELEN),VPTR,VARSIZE)
         VPTR = VPTR + VARSIZE
 300     CONTINUE
C
C       create new variable to hold the sum
      IF (.NOT. SUMOPT) GOTO 400
C
C          initialize the NAMESTR for the new variable
         CALL SAS_XVNAMEI(IADDR(NTYPE))
C
C          fill in NAMESTR information - use DOLLAR15.2 for format
         NTYPE = 1
         NLNG  = 8
         NNAME = CADDR(SUMNAME)
         NAMELEN  = 3
         NLABEL= CADDR(SUMNLABEL)
         NLABLEN  = 24
         NFL   = 15
         NFD   = 2
         NFORM = 'DOLLAR'
         RC = SAS_XVPUTD(OUTPTR,IADDR(NTYPE),0,IADDR(SUM),8)
```

```
C
C        add the SUM element to the variable information array
        RC = SAS_XFNAME(IADDR(NFORM),'F',NUL0,IADDR(VNFCODE))
        VNAMEPTR = NNAME
        VNAMELEN = NAMELEN
        CALL SAS_ZMOVEI(IADDR(VNAMELEN),VARINFOPTR+(NVAR*VARSIZE),
     +                 VARSIZE)
 400    CONTINUE
C
C      complete gather write process
      RC = SAS_XVPUTE(OUTPTR)
C
      OBS=0
      MISSVAL = SAS_ZMISSV('.')
C
      IOBSER = LOCPTR(RBASE,8,OBSERVATPTR)
      IOLD = LOCPTR(RBASE,8,OLDOBSERPTR)
C
C      read and write observations
      IF (SAS_XBYNEXT(INFILE) .LE. 0) GOTO 500
         IF (SAS_XBYGET(INFILE) .EQ. NUL0) GOTO 600
            RC = SAS_XVGET(PTRXV, NUL0)
            SUM = 0
            OBS = OBS + 1
            ALLMISS = .TRUE.
            IF (PRINT)
     +         CALL SAS_ZMOVEI(OBSERVATPTR,OLDOBSERPTR,OBSSIZE)
            DO 700 X=1,NVAR
C
C               handle missing values
               IF (SAS_ZMISS(RBASE(IOBSER+I)) .NE. 1) GOTO 701
                  RBASE(IOBSER+X) = MISSVAL
                  GOTO 702
C
C               otherwise perform calculations
 701           CONTINUE
                  ALLMISS = .FALSE.
                  RBASE(IOBSER+X) = RBASE(IOBSER+X) * MULT
                  IF (SUMOPT)
     +               SUM = SUM + RBASE(IOBSER+X)
 702              CONTINUE

               IF (PRINT)
     +            CALL SAS_ZMOVEI(IADDR(SUM),
     +                            PRINTSUMPTR+(X-1)*8,8)
C
 700           CONTINUE
C
            IF (ALLMISS)
     +         SUM = MISSVAL
C
C             print the observations to the print file
            IF (.NOT. PRINT) GOTO 800
               CALL SAS_XPS('%*t%d', (3+OFFSET), OBS)
               DO 900 X=1,NVAR
                  CALL PRNTOBS(RBASE(IOBSER+X),RBASE(IOLD+X),
     +                         PRINTOPTPTR, VARINFOPTR+(X-1)*VARSIZE)
                  IF (.NOT. SUMOPT) GOTO 901
                     CALL SAS_ZMOVEI(PRINTSUMPTR+(X-1)*8,
     +                               IADDR(TEMP),8)
                     CALL PRNTSUM(TEMP,OFFSET,VARINFOPTR+X*VARSIZE)
 901                 CONTINUE
                  CALL SAS_XPPRN
 900              CONTINUE
               CALL SAS_XPSKIP(1)
 800           CONTINUE
C
C             put the observations out to the new SAS data set
            RC = SAS_XVPUT(OUTPTR,NUL0)
            RC = SAS_XOADD(OUTFILE,NUL0)
 600        CONTINUE
 500     CONTINUE

      RC = SAS_XVPUTT(OUTPTR)
      RC = SAS_XVGETT(PTRXV)
      CALL SAS_XEXIT(XEXITNORMAL,0)
      END
C
C        --Access the variable lists----------------------------------
      SUBROUTINE INITVAR(LIST, NVAR, MULT, INFILE)
      INTEGER*4 LIST,NVAR,INFILE
      REAL*8 MULT
      INTEGER*4 BYLIST,VARLIST,RC
      INCLUDE (PROTOSF)
      INCLUDE (PROC)
```

(continued on next page)

Listing 11.1 (continued)

```
C
C      Creating default variable list if none is given.
      BYLIST = SFLD(PROCHEAD, 0)
      VARLIST = SFLD(PROCHEAD, 4)
      RC = SAS_XVDFLST(INFILE, 1, VARLIST, 1, BYLIST)
      CALL SAS_XPRLOG(RC)
C
C      obtain MULT= option and list of variables
      MULT = SFFD(PROCHEAD, 3, 1.0)
      LIST = SFLP(PROCHEAD, 4)
      NVAR = SFN(PROCHEAD, 4)
      END
C
C     ------ Establish printing environment -----------------
      SUBROUTINE PRTENV(SUM,PRINT,PRINTOPTPTR,INFILE,LIST,NVAR)
      INCLUDE (PROTOSF)
      INCLUDE (PROC)
      LOGICAL*1 SUM,PRINT
      INTEGER*4 PRINTOPTPTR,INFILE,LIST,NVAR
      LOGICAL*1 CENTER
      INTEGER*4 DATE,LABEL,OFFSET,LINESIZE,PAGESIZE
      COMMON/PRINTOP/DATE,LABEL,OFFSET,LINESIZE,PAGESIZE,CENTER
C
      PRINT = (SFOPT(PROCHEAD,1) .NE. 1)
      SUM = (SFOPT(PROCHEAD,2) .NE. 1)
C
      IF (PRINT)
     +   CALL CHSYSOPT(PRINTOPTPTR)
C
C      Load formats for printing
      IF (.NOT. PRINT) GOTO 100
         IF (SAS_XFFILE(INFILE, 'F', LIST, NVAR) .EQ. 0) GOTO 200
            CALL SAS_XPSLOG('%1zCould not load formats.')
            CALL SAS_XEXIT(XEXITERROR, 0)
 200        CONTINUE
C
C         find indention for centering
         OFFSET = FNDOFFS(CENTER,SUM,LINESIZE)
C
         CALL OUTHEAD(SUM,OFFSET)
 100     CONTINUE
C
      END
C
C     ------ CHECK SYSTEM OPTIONS --------------------------------
      SUBROUTINE CHSYSOPT(PRINTOPTPTR)
      INCLUDE (PROTOSF)
      INTEGER*4 PRINTOPTPTR
      INTEGER*4 RC
      LOGICAL*1 CENTER
      INTEGER*4 DATE,LABEL,OFFSET,LINESIZE,PAGESIZE
      COMMON/PRINTOP/DATE,LABEL,OFFSET,LINESIZE,PAGESIZE,CENTER
C
      CENTER = (SAS_XOPTBGT('CENTER', IADDR(RC)) .EQ. 1)
      DATE = SAS_XOPTBGT('DATE', IADDR(RC))
      LABEL = SAS_XOPTBGT('LABEL', IADDR(RC))
      LINESIZE = SAS_XOPTIGT('LINESIZE', IADDR(RC))
      PAGESIZE = SAS_XOPTIGT('PAGESIZE', IADDR(RC))
      END
C
C     ------ Find the offset for printing the print file centered -
      INTEGER FUNCTION FNDOFFS*4(CENTER, SUM, LINESIZE)
      LOGICAL*1 CENTER,SUM
      INTEGER*4 LINESIZE,CNTR,TEXTCNTR,TEXTLEN,OFFSET
C
      TEXTLEN = 54
C
C      revise length of text if the SUM variable should be printed
      IF (SUM)
     +   TEXTLEN = TEXTLEN + 18
C
C      determine the offset if the output should be centered
      IF (.NOT. CENTER) GOTO 100
         CNTR = LINESIZE/2
         TEXTCNTR = TEXTLEN/2
         FNDOFFS = CNTR - TEXTCNTR
         RETURN
 100  FNDOFFS = 0
      RETURN
      END
C
C     ------ Create the headings for the variables ---------------
      SUBROUTINE OUTHEAD(SUM, OFFSET)
      INCLUDE (PROTOSF)
      LOGICAL*1 SUM
      INTEGER*4 OFFSET,LENGTH,I
      CHARACTER*256 HEADER
      CHARACTER*1 X256(256)
      EQUIVALENCE (HEADER,X256)
C
      HEADER = ' '
```

```
C
      I = SAS_XPSSTR(CADDR(HEADER),256,
     +        '%*tOBS %*tVARIABLE %*tPREVIOUS VALUE %*tNEW VALUE',
     +        2+OFFSET,10+OFFSET,22+OFFSET,45+OFFSET)
C
      LENGTH = 54 + OFFSET
      IF (.NOT. SUM) GOTO 100
         I = SAS_XPSSTR(CADDR(X256(I+14)),256-(I+14), 'SUM')
         LENGTH = LENGTH + 18
 100     CONTINUE
C
      CALL SAS_XPHDR(1,CADDR(HEADER),LENGTH)
      CALL SAS_XPAGE
      END
C
C     ----Format the numbers and print them to the print file ------
      SUBROUTINE PRNTOBS(OBSER,OLDOBSER,PRINTOPTPTR,VARINFOPTR)
      INCLUDE (PROTOSF)
      REAL*8 OBSER,OLDOBSER
      INTEGER*4 PRINTOPTPTR
      INTEGER*4 VARINFOPTR
      INTEGER*4 DATE,LABEL,OFFSET,LINESIZE,PAGESIZE
      COMMON/PRINTOP/DATE,LABEL,OFFSET,LINESIZE,PAGESIZE,CENTER
      INTEGER*4 VARSIZE
      COMMON/SIZES/VARSIZE
      INTEGER*4 VNAMELEN,VNAMEPTR,VNFCODE
      COMMON/VARINFO/VNAMELEN,VNAMEPTR,VNFCODE
      CALL SAS_ZMOVEI(VARINFOPTR,IADDR(VNAMELEN),VARSIZE)
      CALL SAS_XPS('%*t%*a',11+OFFSET,VNAMELEN,VNAMEPTR)
      CALL SAS_XPS('%*t%14.*f',22+OFFSET,VNFCODE,OLDOBSER)
      CALL SAS_XPS('%*t%14.*f',40+OFFSET,VNFCODE,OBSER)
      END
C
C     --------- Add the sum to the output if necessary -------------
      SUBROUTINE PRNTSUM(SUM,OFFSET,VARINFOPTR)
      INCLUDE (PROTOSF)
      REAL*8 SUM
      INTEGER*4 VARSIZE
      COMMON/SIZES/VARSIZE
      INTEGER*4 VNAMELEN,VNAMEPTR,VNFCODE
      COMMON/VARINFO/VNAMELEN,VNAMEPTR,VNFCODE
      INTEGER*4 OFFSET
      INTEGER*4 VARINFOPTR
      CALL SAS_ZMOVEI(VARINFOPTR,IADDR(VNAMELEN),VARSIZE)
      CALL SAS_XPS('%*t%15.*f',55+OFFSET,VNFCODE+2,SUM)
      END
```

The MULTIPLY Procedure in the PL/I Language

The sample listing of the MULTIPLY procedure in PL/I follows.

Listing 11.2 *PL/I Version of the MULTIPLY Procedure*

```
%INCLUDE MACPORT6;
 MULTIPLY: PROC PORTMAIN;
 %INCLUDE UWPROC;
 %INCLUDE PROC;
 DCL (ADDR,SUBSTR,CSTG) BUILTIN;
 DCL MULTG ENTRY RETURNS(PTR);

 DCL 1 PRINTOPT BASED(PRINTOPTPTR),
       2 CENTER BIT(1),
       2 (DATE,LABEL,OFFSET) FXL,
       2 (LINESIZE,PAGESIZE) FXL;
 DCL 1 VARINFO(1) BASED(VARINFOPTR),
       2 VNAMELEN FXL,
       2 VNAMEPTR PTR,
       2 VNFCODE FXL;

 DCL
          (INFILE,     /* pointer to the input data set.      */
           OUTFILE,    /* pointer to the output data set.     */
           PTR_XV,     /* pointer for SAS_XVGET calls         */
           OUT_PTR,    /* pointer for SAS_XVPUT calls         */
           PID)        /* pointer to the pool of memory where */
                       /* all the allocated memory comes from.*/
           PTR;
 DCL       LIST(1) FXS /* the list of variable numbers.       */
       BASED(LISTPTR);
 DCL       NVAR FXL;   /* the number of variables in the list.*/
```

(continued on next page)

Listing 11.2 (continued)

```
DCL      (X,I,         /* a counter variable.                  */
          OBS)         /* a counter of the # of observations. */
          FXL;         /*   1= all the variables were missing */
                       /*   an observation.                   */
                       /*   0= there was at least one obs     */
                       /*      that was not missing.          */

DCL      (SUMOPT,      /* a boolean                           */
                       /*      1= print sum in printed output */
                       /*      0= do not print sum in output. */
          PRINT,       /* a boolean                           */
                       /*      1= output to a print file      */
                       /*      0= Do not output to print file.*/
  ALL_MISSING) BIT(1);/* a boolean                            */

DCL      (MULT,        /* the number to multiply by.          */
          SUM)         /* the sum of all the variables for one*/
          FTL;         /* observation.                        */

DCL   OBSERVAT(1) FTL /* pointer to the memory that will     */
   BASED(OBSERVATPTR);/* hold the observations.              */
DCL   OLD_OBSER(1)FTL /* hold the old observations after     */
  BASED(OLD_OBSERPTR);/* multiplying                         */
DCL   PRINT_SUM(1)FTL /* hold the running sum for the        */
  BASED(PRINT_SUMPTR);/* variables in the observation        */

DCL (LISTPTR,OBSERVATPTR,OLD_OBSERPTR,PRINT_SUMPTR,VARINFOPTR,
    PRINTOPTPTR) PTR;
DCL OBSSIZE FXL;
DCL MISSING_VALUE FTL;
DCL NULO PTR;
DCL SUM_NNAME CHAR(8) STATIC INIT('SUM');
DCL SUM_NLABEL CHAR(40) STATIC INIT('SUM OF MULTIPLIED VALUES');

DCL 1 NAMESTR BASED(NAM_PTR),
%INCLUDE NAMESTR;

DCL 1 SUMVAR,
%INCLUDE NAMESTR;

DCL NAM_PTR PTR;            /* a pointer for retrieving the NAMESTR */
                            /* information for variables in the     */
                            /* input data set                       */

CALL UWPRCP(ADDR(PROC));
NULO = SAS_NULL();

/* Parse the PROC Statements */
CALL SAS_XSPARSE(MULTG(),NULO, ADDR(PROC));
IF (PROC.ERROR >= XEXITPROBLEM)
   THEN CALL SAS_XEXIT(XEXITSYNTAX, 0);


CALL SAS_XPSLOG1('SAS/TOOLKIT SOFTWARE ** MULTIPLY PROCEDURE **.');
CALL SAS_XPSLOG1('This procedure is written in PL/I.%n');

/* This call to SAS_XPAGE  is necessary to check any OPTIONS */
/* statements within the PROC call.   */

CALL SAS_XPAGE;

/* Initialize the input and output files */
INFILE = SFFILE(PROC.HEAD, 1);
OUTFILE = SFFILE(PROC.HEAD, 2);

/* Define a pool of memory */
PID = SAS_XMPOOLC(XM_ISA, XM_OSA, XM_EXIT);
PRINTOPTPTR = SAS_XMALLOC(PID,CSTG(PRINTOPT),0);

/* gather information from statement structures and set up the */
/* printing environment based on the options                   */
CALL INITVAR(LISTPTR, NVAR, MULT, INFILE);

CALL PRTENV(SUMOPT, PRINT, PRINTOPTPTR, INFILE, LISTPTR, NVAR);

/* allocate memory for observations */
OBSSIZE = SIZE_FTL * NVAR;
OBSERVATPTR = SAS_XMALLOC(PID, OBSSIZE, 0);
VARINFOPTR = SAS_XMALLOC(PID,CSTG(VARINFO)*(NVAR+1),0);

IF (PRINT) THEN DO;
   OLD_OBSERPTR = SAS_XMALLOC(PID,OBSSIZE,0);
   PRINT_SUMPTR = SAS_XMALLOC(PID,OBSSIZE,0);
   END;

/* define scatter reads */
CALL SAS_XVGETI(INFILE, NVAR, ADDR(PTR_XV));
DO X=1 TO NVAR;
   CALL SAS_XVGETD(PTR_XV,(LIST(X)),0,ADDR(OBSERVAT(X)),SIZE_FTL,
                   XV_NOFMT);
   END;
```

```
CALL SAS_XVGETE(PTR_XV);

/* define gather writes and store info from the NAMESTR  */
CALL SAS_XVPUTI(OUTFILE, NVAR+1, ADDR(OUT_PTR));
DO X=1 TO NVAR;
   CALL SAS_XVNAME(INFILE, (LIST(X)), ADDR(NAM_PTR));
   CALL SAS_XVPUTD(OUT_PTR, NAM_PTR, 0, ADDR(OBSERVAT(X)), SIZE_FTL);
   VARINFO(X).VNAMEPTR = NAM_PTR->NAMESTR.NNAME;
   VARINFO(X).VNAMELEN = NAM_PTR->NAMESTR.NAMELEN;
   VARINFO(X).VNFCODE  = NAM_PTR->NAMESTR.NFCODE;
   END;

/* create new variable to hold the sum */
IF (SUMOPT) THEN DO;

   /* initialize the NAMESTR for the new variable */
   CALL SAS_XVNAMEI(ADDR(SUMVAR));

   /* fill in NAMESTR information - use DOLLAR15.2 for format */
   SUMVAR.NTYPE = 1;
   SUMVAR.NLNG  = SIZE_FTL;
   SUMVAR.NNAME = ADDR(SUM_NNAME);
   SUMVAR.NAMELEN  = 3;
   SUMVAR.NLABEL= ADDR(SUM_NLABEL);
   SUMVAR.NLABLEN  = 24;
   SUMVAR.NFL  = 15;
   SUMVAR.NFD  = 2;
   SUMVAR.NFORM = 'DOLLAR';
   CALL SAS_XVPUTD(OUT_PTR, ADDR(SUMVAR), 0, ADDR(SUM), SIZE_FTL);

   /* add the SUM element to the variable information array */
   CALL SAS_XFNAME(
       ADDR(SUMVAR.NFORM),'F',NUL0,ADDR(VARINFO(NVAR+1).VNFCODE));
   VARINFO(NVAR+1).VNAMEPTR = ADDR(SUMVAR.NNAME);
   VARINFO(NVAR+1).VNAMELEN = SUMVAR.NAMELEN;
   END;

/* complete gather write process */
CALL SAS_XVPUTE(OUT_PTR);

OBS=0;
MISSING_VALUE = SAS_ZMISSV('.');

/* read and write observations */
DO WHILE(SAS_XBYNEXT(INFILE) > 0);
   DO WHILE(SAS_XBYGET(INFILE) ¬= NUL0);
      CALL SAS_XVGET(PTR_XV, NUL0);
      SUM = 0;
      OBS = OBS + 1;
      ALL_MISSING = '1'B;
      IF (PRINT)
         THEN CALL SAS_ZMOVEI(OBSERVATPTR,OLD_OBSERPTR,OBSSIZE);
      DO X=1 TO NVAR;

         /* handle missing values */
         IF (SAS_ZMISS(OBSERVAT(X)) = 1)
            THEN OBSERVAT(X) = MISSING_VALUE;

         /* otherwise perform calculations */
         ELSE DO;
            ALL_MISSING = '0'B;
            OBSERVAT(X) = OBSERVAT(X) * MULT;
            IF (SUMOPT)
               THEN SUM = SUM + OBSERVAT(X);
            END;

         IF (PRINT)
            THEN PRINT_SUM(X) = SUM;

         END;

      IF (ALL_MISSING)
         THEN SUM = MISSING_VALUE;

      /* print the observations to the print file */
      IF (PRINT) THEN DO;
         CALL SAS_XPS3('%*t%d', (3+PRINTOPT.OFFSET), OBS);
         DO X=1 TO NVAR;
            CALL PRNTOBS(OBSERVAT(X),OLD_OBSER(X),PRINTOPTPTR,
                         ADDR(VARINFO(X)));
            IF (SUMOPT)
               THEN CALL PRNTSUM(PRINT_SUM(X), PRINTOPT.OFFSET,
                                 ADDR(VARINFO(NVAR+1)));
            CALL SAS_XPPRN;
            END;
         CALL SAS_XPSKIP(1);
         END;
```

(continued on next page)

Listing 11.2 (continued)

```
         /* put the observations out to the new SAS data set */
         CALL SAS_XVPUT(OUT_PTR, NULO);
         CALL SAS_XOADD(OUTFILE, NULO);
         END;
      END;

   CALL SAS_XVPUTT(OUT_PTR);
   CALL SAS_XVGETT(PTR_XV);
   CALL SAS_XEXIT(XEXITNORMAL, 0); /* No return or stop */


     /*--Access the variable lists-----------------------------------*/
   INITVAR: PROC(LIST, NVAR, MULT, INFILE) PORTPROC;
   DCL LIST PTR;
   DCL NVAR FXL;
   DCL MULT FTL;
   DCL INFILE PTR;
   DCL (BYLIST,VARLIST) PTR;
   DCL (RC,ONE) FXL;

   /* Creating default variable list if none is given. */
   BYLIST = SFLD(PROC.HEAD, 0);
   VARLIST = SFLD(PROC.HEAD, 4);
   ONE = 1;
   RC = SAS_XVDFLST(INFILE, ONE, VARLIST, ONE, BYLIST);
   CALL SAS_XPRLOG(RC);

   /* obtain MULT= option and list of variables */
   MULT = SFFD(PROC.HEAD, 3, 1.0);
   LIST = SFLP(PROC.HEAD, 4);
   NVAR = SFN(PROC.HEAD, 4);
   END INITVAR;

   /*------ Establish printing environment ------------------*/
   PRTENV: PROC(SUM, PRINT, PRINTOPTPTR, INFILE, LIST, NVAR) PORTPROC;
   DCL SUM BIT(1);
   DCL PRINT BIT(1);
   DCL PRINTOPTPTR PTR;
   DCL INFILE PTR;
   DCL LIST PTR;
   DCL NVAR FXL;

   PRINT = (SFOPT(PROC.HEAD,1) ¬= 1);
   SUM = (SFOPT(PROC.HEAD,2) ¬= 1);

   IF (PRINT)
      THEN CALL CHSYSOPT(PRINTOPTPTR);

   /* Load formats for printing */
   IF (PRINT) THEN DO;
      IF (SAS_XFFILE(INFILE, 'F', LIST, NVAR)) THEN DO;
         CALL SAS_XPSLOG1('%1zCould not load formats.');
         CALL SAS_XEXIT(XEXITERROR, 0);
         END;

      /* find indention for centering */
      PRINTOPTPTR->OFFSET = FNDOFFS(PRINTOPTPTR->CENTER, SUM,
                            PRINTOPTPTR->LINESIZE);

      CALL OUTHEAD(SUM, PRINTOPTPTR->OFFSET);
      END;

   END PRTENV;

   /*------ CHECK SYSTEM OPTIONS ----------------------------------*/
   CHSYSOPT: PROC(PRINTOPTPTR) PORTPROC;
   DCL PRINTOPTPTR PTR;
   DCL RC FXL;

   PRINTOPTPTR->CENTER   = (SAS_XOPTBGT('CENTER', ADDR(RC)) = 1);
   PRINTOPTPTR->DATE     = SAS_XOPTBGT('DATE', ADDR(RC));
   PRINTOPTPTR->LABEL    = SAS_XOPTBGT('LABEL', ADDR(RC));
   PRINTOPTPTR->LINESIZE = SAS_XOPTIGT('LINESIZE', ADDR(RC));
   PRINTOPTPTR->PAGESIZE = SAS_XOPTIGT('PAGESIZE', ADDR(RC));
   END CHSYSOPT;

   /*------ Find the offset for printing the print file centered -*/
   FNDOFFS: PROC(CENTER, SUM, LINESIZE) PORTPROC RETURNS(FXL);
   DCL CENTER BIT(1);
   DCL SUM BIT(1);
   DCL LINESIZE FXL;
   DCL (CNTR,TEXTCNTR,TEXTLEN,OFFSET) FXL;

   TEXTLEN = 54;

   /* revise length of text if the SUM variable should be printed */
   IF (SUM)
      THEN TEXTLEN = TEXTLEN + 18;
```

```
/* determine the offset if the output should be centered */
IF (CENTER) THEN DO;
   CNTR = LINESIZE/2;
   TEXTCNTR = TEXTLEN/2;
   OFFSET = CNTR - TEXTCNTR;
   RETURN(OFFSET);
   END;
RETURN(0);
END FNDOFFS;

/*------ Create the headings for the variables ----------------*/
OUTHEAD: PROC(SUM, OFFSET) PORTPROC;
DCL SUM BIT(1);
DCL OFFSET FXL;
DCL HEADER CHAR(256);
DCL X256(256) CHAR(1) BASED(ADDR(HEADER));
DCL LENGTH FXL;
DCL I FXL;

HEADER = ' ';

I = SAS_XPSSTR7(ADDR(HEADER),256,
                '%*tOBS %*tVARIABLE %*tPREVIOUS VALUE %*tNEW VALUE',
                (2+OFFSET), (10+OFFSET), (22+OFFSET), (45+OFFSET));

LENGTH = 54 + OFFSET;
IF (SUM) THEN DO;
   I = SAS_XPSSTR3(ADDR(X256(I+14)),256-(I+14), 'SUM');
   LENGTH = LENGTH + 18;
   END;

CALL SAS_XPHDR(1, ADDR(HEADER), LENGTH);
CALL SAS_XPAGE;
END;

/*----Format the numbers and print them to the print file ------*/
PRNTOBS: PROC(OBSER, OLD_OBSER, PRINTOPTPTR, VARINFOPTR) PORTPROC;
DCL (OBSER,OLD_OBSER) FTL;
DCL PRINTOPTPTR PTR;
DCL VARINFOPTR PTR;
DCL 1 VARINFO BASED(VARINFOPTR),
      2 VNAMELEN FXL,
      2 VNAMEPTR PTR,
      2 VNFCODE FXL;
CALL SAS_XPS4('%*t%*a', (11+PRINTOPTPTR->OFFSET),
          VARINFOPTR->VNAMELEN,
          VARINFOPTR->VNAMEPTR);
CALL SAS_XPS4('%*t%14.*f', (22+PRINTOPTPTR->OFFSET),
          VARINFOPTR->VNFCODE,
          OLD_OBSER);
CALL SAS_XPS4('%*t%14.*f', (40+PRINTOPTPTR->OFFSET),
          VARINFOPTR->VNFCODE, OBSER);
END PRNTOBS;

/*--------- Add the sum to the output if necessary -------------*/
PRNTSUM: PROC(SUM, OFFSET, VARINFOPTR) PORTPROC;
DCL SUM FTL;
DCL OFFSET FXL;
DCL VARINFOPTR PTR;
DCL 1 VARINFO BASED(VARINFOPTR),
      2 VNAMELEN FXL,
      2 VNAMEPTR PTR,
      2 VNFCODE FXL;
CALL SAS_XPS4('%*t%15.*f', (55+OFFSET), VARINFOPTR->VNFCODE+2, SUM);
END PRNTSUM;

END MULTIPLY;
```

The MULTIPLY Procedure in the IBM 370 Assembler Language

The sample listing of the MULTIPLY procedure in the IBM 370 assembler language follows.

Listing 11.3
IBM 370 Assembler Language Version of the MULTIPLY Procedure

```
         COPY  UWPROC
MAIN     CSECT
         USING MAIN,R9
         USING MAIN+X'1000',R10
         USING MAIN+X'2000',R11
         USING MAIND,R8
*--------Prologue-------------------------------------------------------*
         STM   R14,R12,12(R13)        Save caller's regs
         LR    R9,R15                 First base reg
         LA    R11,4095
         LA    R10,1(R11,R9)          Second base reg
         LA    R11,1(R11,R10)         Third base reg
         LA    R8,NONRENT             Addressability to DSECT
         LA    R2,SAVEAREA
         ST    R2,8(R13)              Save our sa ptr in caller's sa
         ST    R13,4(R2)              Save caller's sa ptr in our sa
         LR    R13,R2                 Now use our sa ptr
*
**************************INITIALIZATION*****************************
*
*-------------------------------------------------------------------*
* UWPRCC is called to initialize the SAS/Toolkit SAS procedure      *
* environment. This is a required call, and should be made exactly  *
* once, and should be the first executable statement.               *
*-------------------------------------------------------------------*
*
         LA    R1,PROCSTR
         ST    R1,PARMLIST
         LA    R1,PARMLIST
         L     R15,=V(UWPRCC)
         BALR  R14,R15
*
*-------------------------------------------------------------------*
* We call the grammar function to get the pointer that will be      *
* needed by SAS_XSPARSE.                                            *
*-------------------------------------------------------------------*
*
         SLR   R1,R1
         L     R15,=V(MULTG)
         BALR  R14,R15
*
*-------------------------------------------------------------------*
* SAS_XSPARSE should be called with the grammar function's returned *
* pointer, a NULL, and the proc structure address. The proc.error   *
* value will be set to indicate parsing success. If it indicates    *
* that parsing problems occur, we simply abandon ship.              *
*-------------------------------------------------------------------*
*
*     RC = SAS_XSPARSE(MULTG(),NULO,IADDR(PROCHEAD))
*
         SLR   R0,R0
         LA    R1,PROCSTR
         STM   R15,R1,PARMLIST
         LA    R1,PARMLIST
         L     R15,&SAS_XSPARSE
         BALR  R14,R15
*
*     IF (PROC_ERROR > XEXITPROBLEM)
*        CALL XEXIT(PROC_ERROR,0)
         L     R0,PROCSTR+PROC_ERROR-PROC
         C     R0,=A(XEXITPROBLEM)
         BL    MESSAGE1
         SLR   R1,R1
         STM   R0,R1,PARMLIST
         LA    R1,PARMLIST
         L     R15,&SAS_XEXIT
         BALR  R14,R15
MESSAGE1 DS    0H
         LOGLIT1 'SAS/TOOLKIT SOFTWARE ** MULTIPLY PROCEDURE **.'
         LA    R1,PARMLIST
         L     R15,&SAS_XPSLOG
         BALR  R14,R15
         LOGLIT1 'This procedure is written in IBM Assembler.'
         LA    R1,PARMLIST
         L     R15,&SAS_XPSLOG
         BALR  R14,R15
*
         SLR   R1,R1
         L     R15,&SAS_XPAGE
         BALR  R14,R15
*
*     /* Initialize the input and output files */
         SFFILE PROC_HEAD-PROC+PROCSTR,=F'1'
         ST    R1,INFILE
         SFFILE PROC_HEAD-PROC+PROCSTR,=F'2'
         ST    R1,OUTFILE
*
```

```
*     /* gather information from statement structures and set up the */
*     /* printing environment based on the options                    */
*
*  Init_var(&list, &nvar, &mult, infile);
*
         LA    R14,LIST
         LA    R15,NVAR
         LA    R0,MULT
         L     R1,INFILE
         STM   R14,R1,PARMLIST
         LA    R1,PARMLIST
         L     R15,=A(INITVAR)
         BALR  R14,R15
*
*  Prt_env(&SUM, &PRINT, &printopt, infile, list, nvar);
         LA    R14,SUMOPT
         LA    R15,PRINT
         LA    R0,PRINTOPTC
         L     R1,INFILE
         L     R2,LIST
         LH    R3,NVAR
         STM   R14,R3,PARMLIST
         LA    R1,PARMLIST
         L     R15,=A(PRTENV)
         BALR  R14,R15
*
*     /* Define a pool of memory */
*  pid = SAS_XMPOOLC(XM_ISA, XM_OSA, XM_EXIT);
*
         L     R14,=A(XM_ISA)
         L     R15,=A(XM_OSA)
         L     R0,=A(XM_EXIT)
         STM   R14,R0,PARMLIST
         LA    R1,PARMLIST
         L     R15,&SAS_XMPOOLC
         BALR  R14,R15
         ST    R15,PID
*
*     /* allocate memory for observations */
*  observat = (double *)SAS_XMALLOC(pid, sizeof(double)*nvar, 0);
*
         L     R14,PID
         LH    R15,NVAR
         SLA   R15,3
         SLR   R0,R0
         STM   R14,R0,PARMLIST
         LA    R1,PARMLIST
         L     R15,&SAS_XMALLOC
         BALR  R14,R15
         ST    R15,OBSERVAT
*
*  varinfo = (struct UWINFOSTR *)SAS_XMALLOC(pid,
*           (sizeof(struct UWINFOSTR))*(nvar+1), 0);
*
*  first parmlist element already PID
         LH    R15,NVAR
         LA    R15,1(R15)
         LA    R14,VARINFOL
         MR    R14,R15
         ST    R15,PARMLIST+4
*  third parmlist element already 0 and R1 already points to parmlist
         L     R15,&SAS_XMALLOC
         BALR  R14,R15
         ST    R15,VARINFOPTR
*
*  if (PRINT) {
*     old_obser = (double *)SAS_XMALLOC(pid, sizeof(double)*nvar, 0);
*     print_sum = (double *)SAS_XMALLOC(pid, sizeof(double)*nvar, 0);
*  }
         CLI   PRINT,0
         BE    X00001
         LH    R15,NVAR
         SLA   R15,3
         ST    R15,PARMLIST+4
         L     R15,&SAS_XMALLOC
         BALR  R14,R15
         ST    R15,OLDOBSER
         L     R15,&SAS_XMALLOC
         BALR  R14,R15
         ST    R15,PRINTSUM
X00001   DS    0H
```

(continued on next page)

Listing 11.3
(continued)

```
*
*     /* define scatter reads */
*   SAS_XVGETI(infile, nvar, &ptr_xv);
*
          L     R14,INFILE
          LH    R15,NVAR
          LA    R0,PTRXV
          STM   R14,R0,PARMLIST
          L     R15,&SAS_XVGETI
          BALR  R14,R15
*
*   short *r5; dblptr r4;
*   for(x=nvar,r5=list,r4=observat;x>0;x--,r5++,r4++) {
*      SAS_XVGETD(ptr_xv,*r5, 0, r4, sizeof(double),XV_NOFMT);
*      }
          L     R14,PTRXV
          LA    R2,8
          LA    R3,XV_NOFMT
          STM   R14,R3,PARMLIST
          L     R5,LIST
          SLR   R3,R3
          L     R4,OBSERVAT
          LH    R0,NVAR
          LTR   R0,R0
          BZ    ENDLOOP1
NEXTCAL1 DS     0H
          LH    R2,0(R5)
          STM   R2,R4,PARMLIST+4
          L     R15,&SAS_XVGETD
          BALR  R14,R15
          LA    R5,2(R5)
          LA    R4,8(R4)
          BCT   R0,NEXTCAL1
ENDLOOP1 DS     0H
*
*   SAS_XVGETE(ptr_xv);
*
          L     R15,&SAS_XVGETE
          BALR  R14,R15
*
*     /* define gather writes and store info from the NAMESTR   */
*   SAS_XVPUTI(outfile, nvar+1, &out_ptr);
*
          L     R14,OUTFILE
          LH    R15,NVAR
          LA    R15,1(R15)
          LA    R0,OUTPTR
          STM   R14,R0,PARMLIST
          LA    R1,PARMLIST
          L     R15,&SAS_XVPUTI
          BALR  R14,R15
*
*   for(x=0; x<nvar; x++) {
*      SAS_XVNAME(infile, list(|x|), &Nam_ptr);
*      SAS_XVPUTD(out_ptr, Nam_ptr, 0, observat+x, sizeof(double));
*      memcpy(varinfo(|x|).name, Nam_ptr->nname, Nam_ptr->namelen);
*      varinfo(|x|).namelen = Nam_ptr->namelen;
*      varinfo(|x|).nfcode = Nam_ptr->nfcode;
*   }
*
          L     R3,LIST
          L     R4,OBSERVAT
          L     R5,VARINFOPTR
          SLR   R1,R1
TOPLOOP2 DS     0H
          CH    R1,NVAR
          BNL   ENDLOOP2
          ST    R1,X
          L     R14,INFILE
          LH    R15,0(R3)
          LA    R0,NAMPTR
          STM   R14,R0,PARMLIST
          LA    R1,PARMLIST
          L     R15,&SAS_XVNAME
          BALR  R14,R15
          L     R14,OUTPTR
          L     R15,NAMPTR
          SLR   R0,R0
          LR    R1,R4
          LA    R2,8
          STM   R14,R2,PARMLIST
          LA    R1,PARMLIST
          L     R15,&SAS_XVPUTD
          BALR  R14,R15
          L     R2,NAMPTR
          MVC   VNAMEPTR-VARINFO(4,R5),NNAME-NAMESTR(R2)
          MVC   VNAMELEN-VARINFO(4,R5),NAMELEN-NAMESTR(R2)
          MVC   VNFCODE-VARINFO(4,R5),NFCODE-NAMESTR(R2)
          LA    R3,2(R3)
          LA    R4,8(R4)
          LA    R5,VARINFOL(R5)
          L     R1,X
```

```
         LA    R1,1(R1)
         B     TOPLOOP2
ENDLOOP2 DS    0H
*
*    /* create new variable to hold the sum */
*  if (SUM) {
         CLI   SUMOPT,0
         BE    X0002
*
*        /* initialize the NAMESTR for the new variable */
*     SAS_XVNAMEI(&sumvar);
         LA    R2,SUMVAR
         ST    R2,PARMLIST
         LA    R1,PARMLIST
         L     R15,&SAS_XVNAMEI
         BALR  R14,R15
*
*        /* fill in NAMESTR information - use DOLLAR15.2 for format */
*     sumvar.ntype = 1;
*     sumvar.nlng  = sizeof(double);
*     sumvar.nname="SUM";
*     sumvar.namelen  = 3;
*     sumvar.nlabel="Sum of multiplied Values";
*     sumvar.nlablen  = 24;
*     sumvar.nfl  = 15;
*     sumvar.nfd  = 2;
*     memcpy(sumvar.nform, "DOLLAR  ", 8);
         MVC   NTYPE-NAMESTR(4,R2),=F'1'
         MVC   NLNG-NAMESTR(4,R2),=F'8'
         LA    R0,SUM_LIT
         ST    R0,NNAME-NAMESTR(R2)
         MVC   NAMELEN-NAMESTR(4,R2),=F'3'
         LA    R0,=CL3'Sum of multiplied values'
         ST    R0,NLABEL-NAMESTR(R2)
         MVC   NLABLEN-NAMESTR(4,R2),=F'24'
         MVC   NFL-NAMESTR(4,R2),=F'15'
         MVC   NFD-NAMESTR(4,R2),=F'2'
         MVC   NFORM-NAMESTR(8,R2),=CL8'DOLLAR'
*
*     SAS_XVPUTD(out_ptr, &sumvar, 0, &sum, sizeof(double));
*
         L     R14,OUTPTR
         LR    R15,R2
         SLR   R0,R0
         LA    R1,SUM
         LA    R2,8
         STM   R14,R2,PARMLIST
         LA    R1,PARMLIST
         L     R15,&SAS_XVPUTD
         BALR  R14,R15
*
*       /* add the SUM element to the variable information array */
*     SAS_XFNAME("DOLLAR", 'F', NULL, &(varinfo(|nvar|).nfcode));
*
         LA    R14,=CL8'DOLLAR'
         LA    R15,C'F'
         SLR   R0,R0
         LA    R2,VARINFOL
         MH    R2,NVAR
         L     R1,VARINFOPTR
         LA    R2,0(R1,R2)
         LA    R1,VNFCODE-VARINFO(R2)
         STM   R14,R1,PARMLIST
         LA    R1,PARMLIST
         L     R15,&SAS_XFNAME
         BALR  R14,R15
*
*     memcpy(varinfo(|nvar|).name, sumvar.nname, 8);
*     varinfo(|nvar|).namelen = sumvar.namelen;
         MVC   VNAMEPTR-VARINFO(4,R2),NNAME-NAMESTR+SUMVAR
         MVC   VNAMELEN-VARINFO(4,R2),NAMELEN-NAMESTR+SUMVAR
*
X0002    DS    0H
*
*    /* complete gather write process */
*
*  SAS_XVPUTE(out_ptr);
*
         MVC   PARMLIST(4),OUTPTR
         LA    R1,PARMLIST
         L     R15,&SAS_XVPUTE
         BALR  R14,R15
```

(continued on next page)

Listing 11.3
(continued)

```
*
*  obs=0;
         XC     OBS(4),OBS
         L      R1,INFILE
         ST     R1,XBYPARM
         L      R0,PTRXV
         SLR    R1,R1
         STM    R0,R1,XVGETPRM
         L      R0,OUTPTR
         STM    R0,R1,XVPUTPRM
         L      R0,OUTFILE
         STM    R0,R1,XOADDPRM
*
*     /* read and write observations */
*  while(SAS_XBYNEXT(infile) > 0) {
*
X0003    DS     0H
         LA     R1,XBYPARM
         L      R15,&SAS_XBYNEXT
         BALR   R14,R15
         LTR    R15,R15
         BNP    X0004
*
*     while(SAS_XBYGET(infile) != NULL) {
*
X0005    DS     0H
         LA     R1,XBYPARM
         L      R15,&SAS_XBYGET
         BALR   R14,R15
         LTR    R15,R15
         BZ     X0012
*
*        SAS_XVGET(ptr_xv, NULL);
*
         LA     R1,XVGETPRM
         L      R15,&SAS_XVGET
         BALR   R14,R15
*
*        sum=0;
*        obs++;
*        all_missing = 1;
*
         XC     SUM(8),SUM
         L      R1,OBS
         LA     R1,1(R1)
         ST     R1,OBS
         MVI    ALLMISS,1
*
*        for(x=0;x<nvar;x++) {
*
         LA     R1,PARMLIST
         L      R2,OLDOBSER
         L      R3,OBSERVAT
         L      R4,PRINTSUM
         LH     R0,NVAR
         LTR    R0,R0
         BZ     X0006
*
*           if (PRINT) old_obser(|x|) = observat(|x|);
*              /* handle missing values */
*
X0007    DS     0H
         CLI    PRINT,0
         BE     *+4+6
         MVC    0(8,R2),0(R3)
*
*           if (SAS_ZMISS(observat(|x|))) {
*              observat(|x|) = MACMISSING;
*              }
*
         MVC    PARMLIST(8),0(R3)
         L      R15,&SAS_ZMISS
         BALR   R14,R15
         LTR    R15,R15
         BZ     *+4+6+4
         MVC    0(8,R3),&MACMISSING
         B      X0008
```

```
*
*             /* otherwise perform calculations */
*           else {
*             all_missing = 0;
*             observat(|x|) *= mult;
*             if (SUMOPT) sum += observat(|x|);
*             }
*
         MVI   ALLMISS,0
         LD    FR0,0(R3)
         MD    FR0,MULT
         STD   FR0,0(R3)
         CLI   SUMOPT,0
         BE    *+4+4+4
         AD    FR0,SUM
         STD   FR0,SUM
X0008    DS    0H
*
*           if (PRINT) print_sum(|x|) = sum;
*
         CLI   PRINT,0
         BE    *+4+6
         MVC   0(8,R4),SUM
*
         LA    R2,8(R2)
         LA    R3,8(R3)
         LA    R4,8(R4)
         BCT   R0,X0007
*
X0006    DS    0H
*
*           if (all_missing) sum = MACMISSING;
*
         CLI   ALLMISS,1
         BNE   *+4+6
         MVC   SUM(8),&MACMISSING
*
*           /* print the observations to the print file */
*        if (PRINT) {
*
         CLI   PRINT,0
         BE    X0009
*
*           SAS_XPS("%*t%d", (3+printopt.offset), obs);
*
         LOGLIT1 '%*t%d'
         L     R1,OFFSET-PRINTOPT+PRINTOPTC
         AH    R1,=H'3'
         L     R2,OBS
         STM   R1,R2,PARMLIST+4
         LA    R1,PARMLIST
         L     R15,&SAS_XPS
         BALR  R14,R15
*
*           for(x=0;x<nvar;x++) {
*               Prnt_obs(observat(|x|), old_obser(|x|),&printopt,
*                        &varinfo(|x|));
*               if (SUM) Prnt_sum(print_sum(|x|), printopt.offset,
*                                 &varinfo(|nvar|));
*               SAS_XPPRN();
*               }
*
         LA    R1,PARMLIST
         L     R2,OBSERVAT
         L     R3,OLDOBSER
         L     R4,VARINFOPTR
         L     R5,PRINTSUM
         LH    R6,NVAR
         MH    R6,VARINFOL
         LA    R6,0(R4,R6)
         SLR   R15,R15
```

(continued on next page)

Listing 11.3
(continued)

```
X0010    DS    0H
         ST    R15,X
         CH    R15,NVAR
         BNL   X0011
         MVC   PARMLIST(8),0(R2)
         MVC   PARMLIST+8(8),0(R3)
         LA    R14,PRINTOPTC
         LR    R15,R4
         STM   R14,R15,PARMLIST+16
         L     R15,=A(PRNTOBS)
         BALR  R14,R15
         CLI   SUMOPT,0
         BE    CALXPRN
         MVC   PARMLIST(8),0(R5)
         L     R14,OFFSET-PRINTOPT+PRINTOPTC
         LR    R15,R6
         STM   R14,R15,PARMLIST+8
         L     R15,=A(PRNTSUM)
         BALR  R14,R15
CALXPRN  L     R15,&SAS_XPPRN
         BALR  R14,R15
         LA    R2,8(R2)
         LA    R3,8(R3)
         LA    R4,VARINFOL(R4)
         LA    R5,8(R5)
         L     R15,X
         LA    R15,1(R15)
         B     X0010
X0011    DS    0H
*
*           SAS_XPSKIP(1);
*
         MVC   PARMLIST(4),=F'1'
         L     R15,&SAS_XPSKIP
         BALR  R14,R15
*
*           /* put the observations out to the new SAS data set */
*        SAS_XVPUT(out_ptr, NULL);
*
X0009    DS    0H
         LA    R1,XVPUTPRM
         L     R15,&SAS_XVPUT
         BALR  R14,R15
*
*        SAS_XOADD(outfile, NULL);
*
         LA    R1,XOADDPRM
         L     R15,&SAS_XOADD
         BALR  R14,R15
         B     X0005
X0012    DS    0H                                   END OF SAS_XBYGET LOOP
         B     X0003
X0004    DS    0H                                   END OF SAS_XBYNEXT LOOP
*
*  SAS_XVPUTT(out_ptr);
*  SAS_XVGETT(ptr_xv);
*
         LA    R1,XVPUTPRM
         L     R15,&SAS_XVPUTT
         BALR  R14,R15
         LA    R1,XVGETPRM
         L     R15,&SAS_XVGETT
         BALR  R14,R15
*
*  SAS_XEXIT(XEXITNORMAL, 0); /* No return or stop */
*
         L     R15,=A(XEXITNORMAL)
         SLR   R0,R0
         STM   R15,R0,PARMLIST
         LA    R1,PARMLIST
         L     R15,&SAS_XEXIT
         BALR  R14,R15
*
*
* void    Init_var(list, nvar, mult, infile)
* short   **list,
*         *nvar;
* double  *mult;
* ptr     infile;
*
INITVAR  DS    0H
         STM   R14,R7,INITVAR#
         MVC   INITPARM(16),0(R1)
```

```
*
*     /* Creating default variable list if none is given. */
*  bylist =(struct STLIST *)SFLD(proc.head, 0);
*  varlist =(struct STLIST *)SFLD(proc.head, 4);
*  rc = SAS_XVDFLST(infile, 1, varlist, 1, bylist);
*  SAS_XPRLOG(rc);
*
         SFLD  PROC_HEAD-PROC+PROCSTR,=F'0'
         ST    R1,BYLIST_I
         SFLD  PROC_HEAD-PROC+PROCSTR,=F'4'
         ST    R1,VARLIST_I
         L     R14,INITPARM+3*4
         LA    R15,1
         L     R0,VARLIST_I
         LR    R1,R15
         L     R2,BYLIST_I
         STM   R14,R2,PARMLIST
         LA    R1,PARMLIST
         L     R15,$SAS_XVDFLST
         BALR  R14,R15
         ST    R15,PARMLIST
         L     R15,$SAS_XPRLOG
         BALR  R14,R15
*
*     /* obtain MULT= option and list of variables */
*  if (SFTYPE(proc.head, 3)) *mult = SFF(proc.head, 3);
*  *list = (short *)SFLP(proc.head, 4);
*  *nvar = SFN(proc.head, 4);
*
         SFFD  PROC_HEAD-PROC+PROCSTR,=F'3',=D'1'
         L     R3,INITPARM+2*4
         STD   FR0,0(R3)
         SFLP  PROC_HEAD-PROC+PROCSTR,=F'4'
         L     R3,INITPARM+0*4
         ST    R1,0(R3)
         SFN   PROC_HEAD-PROC+PROCSTR,=F'4'
         L     R3,INITPARM+1*4
         STH   R1,0(R3)
         L     R14,INITVAR#
         LM    R0,R7,INITVAR#+8
         BR    R14
         SPACE 3
MAIND    DSECT
INITVAR# DS    10F
INITPARM DS    16C
BYLIST_I DS    A
VARLIST_I DS   A
         EJECT
MAIN     CSECT
*
*  /*------ Establish printing environment -----------------*/
*  void    Prt_env(SUMOPT, PRINT, printopt, infile, list, nvar)
*  int     *SUMOPT,
*          *PRINT;
*  struct SYSOPTS *printopt;
*  ptr     infile;
*
PRTENV   DS    0H
         STM   R14,R7,PRTENV#
         MVC   PRTEPARM(6*4),0(R1)
*
*  *SUM = SFOPT(proc.head, 2) ? 0 : 1;
*  *PRINT = SFOPT(proc.head, 1) ? 0 : 1;
*
         SFO   PROC_HEAD-PROC+PROCSTR,=F'2'
         X     R0,=F'1'
         L     R1,PRTEPARM+0*4
         STC   R0,0(R1)
         SFO   PROC_HEAD-PROC+PROCSTR,=F'1'
         X     R0,=F'1'
         L     R2,PRTEPARM+1*4
         STC   R0,0(R2)
*
*  if (*PRINT) {
*     Chsysopt(printopt);
*
         CLI   0(R1),0
         BE    CALXFILE
         MVC   PARMLIST(4),PRTEPARM+2*4
         LA    R1,PARMLIST
         L     R15,=A(CHSYSOPT)
         BALR  R14,R15
```

(continued on next page)

Listing 11.3 (continued)

```
*
*      /* Load formats for printing */
*      if (SAS_XFFILE(infile, 'F', list, nvar)) {
*         SAS_XPSLOG("%1zCould not load format.");
*         SAS_XEXIT(XEXITERROR, 0);
*      }
*
CALXFILE DS    0H
         L     R14,PRTEPARM+3*4
         LA    R15,C'F'
         LM    R0,R1,PRTEPARM+4*4
         STM   R14,R1,PARMLIST
         LA    R1,PARMLIST
         L     R15,&SAS_XFFILE
         BALR  R14,R15
         LTR   R15,R15
         BZ    CALLFNDO
         LOGLIT1 '%1zCould not load formats.'
         LA    R1,PARMLIST
         L     R15,&SAS_XPSLOG
         BALR  R14,R15
         L     R15,=A(XEXITERROR)
         SLR   R0,R0
         STM   R15,R0,PARMLIST
         LA    R1,PARMLIST
         L     R15,&SAS_XEXIT
         BALR  R14,R15
*
*        /* find indention for centering */
*      printopt->offset = Fnd_offs(printopt->center, *SUMOPT,
*                          printopt->linesize);
*
CALLFNDO DS    0H
         L     R2,PRTEPARM+2*4
         L     R14,CENTER-PRINTOPT(R2)
         L     R15,PRTEPARM+0*8
         SLR   R0,R0
         IC    R0,0(R15)
         LR    R15,R0
         ST    R15,SUMOPTX
         L     R0,LINESIZE-PRINTOPT(R2)
         STM   R14,R0,PARMLIST
         LA    R1,PARMLIST
         L     R15,=A(FNDOFFS)
         BALR  R14,R15
         ST    R15,OFFSET-PRINTOPT(R2)
*
*      Out_head(*SUMOPT, printopt->offset);
*
         L     R14,SUMOPTX
         STM   R14,R15,PARMLIST
         LA    R1,PARMLIST
         L     R15,=A(OUTHEAD)
         BALR  R14,R15
*
         L     R14,PRTENV#
         LM    R0,R7,PRTENV#+8
         BR    R14
         SPACE 3
MAIND    DSECT
PRTENV#  DS    10F
PRTEPARM DS    24C
SUMOPTX  DS    F
MAIN     CSECT
         EJECT
*
CHSYSOPT DS    0H
*
*   /*------ Check system Options ---------------------------------*/
*   void    Chsysopt(printopt)
*
         STM   R14,R7,CHSYSOP#
         L     R2,0(R1)
         LA    R1,RC_C
         ST    R1,PARMLIST+4
         LA    R1,PARMLIST
*
*   printopt->center    = SAS_XOPTBGT("CENTER", &rc);
*
         LOGLIT1 'CENTER'
         L     R15,&SAS_XOPTBGT
         LA    R1,PARMLIST
         BALR  R14,R15
         ST    R15,CENTER-PRINTOPT(R2)
```

```
*
*  printopt->date     = SAS_XOPTBGT("DATE", &rc);
*
         LOGLIT1 'DATE'
         L     R15,&SAS_XOPTBGT
         LA    R1,PARMLIST
         BALR  R14,R15
         ST    R15,DATE-PRINTOPT(R2)
*
*  printopt->label    = SAS_XOPTBGT("LABEL", &rc);
*
         LOGLIT1 'LABEL'
         L     R15,&SAS_XOPTBGT
         LA    R1,PARMLIST
         BALR  R14,R15
         ST    R15,PLABEL-PRINTOPT(R2)
*
*  printopt->linesize = SAS_XOPTIGT("LINESIZE", &rc);
*
         LOGLIT1 'LINESIZE'
         L     R15,&SAS_XOPTIGT
         LA    R1,PARMLIST
         BALR  R14,R15
         ST    R15,LINESIZE-PRINTOPT(R2)
*
*  printopt->pagesize = SAS_XOPTIGT("PAGESIZE", &rc);
*
         LOGLIT1 'PAGESIZE'
         L     R15,&SAS_XOPTIGT
         LA    R1,PARMLIST
         BALR  R14,R15
         ST    R15,PAGESIZE-PRINTOPT(R2)
*
         L     R14,CHSYSOP#
         LM    R0,R7,CHSYSOP#+8
         BR    R14
         SPACE 3
MAIND    DSECT
CHSYSOP# DS    10F
RC_C     DS    F
MAIN     CSECT
         EJECT
*
FNDOFFS  DS    0H
*
*     /*------ Find the offset for printing the print file centered -*/
*  int      Fnd_offs(center, SUMOPT, linesize)
*  int      center,SUMOPT;
*  long     linesize;
*
*  int      cntr,
*           textcntr,
*           textlen = 54,
*           offset;
*
         STM   R14,R7,FNDOFFS#
         SLR   R15,R15
         MVC   FNDOPARM(12),0(R1)
         LA    R3,54
*
*     /* revise length of text if the SUM variable should be printed */
*  if (SUM) textlen +=18;
*
         L     R1,FNDOPARM+1*4
         LTR   R1,R1
         BZ    *+4+4
         LA    R3,18(R3)
*
*     /* determine the offset if the output should be centered */
*  if (center) {
*
         L     R1,FNDOPARM+0*4
         LTR   R1,R1
         BZ    RET0
```

(continued on next page)

Listing 11.3
(continued)

```
*
*     cntr = linesize/2;
*     textcntr = textlen/2;
*     offset = cntr - textcntr;
*     return(offset);
*     }
*
         L     R15,FNDOPARM+2*4
         SRA   R15,1
         SRA   R3,1
         SLR   R15,R3
*
*  return(0);
*
RET0     L     R14,FNDOFFS#
         LM    R0,R7,FNDOFFS#+8
         BR    R14
         SPACE 3
MAIND    DSECT
FNDOFFS# DS    10F
FNDOPARM DS    12C
MAIN     CSECT
         EJECT
OUTHEAD  DS    0H
*
*  /*------ Create the headings for the variables ----------------*/
* void    Out_head(SUMOPT, offset)
* int     SUMOPT,offset;
*
*  char header(|256|);
*  int  length,i;
*
         STM   R14,R7,OUTHEAD#
         MVC   OUTHPARM(8),0(R1)
*
*  SAS_ZFILLCI(' ', header, 256);
*
         MVI   HEADER_O,C' '
         MVC   HEADER_O+1(255),HEADER_O
*
*  i = SAS_XPSSTR(header,256,
*                 "%*tOBS %*tVARIABLE %*tPREVIOUS VALUE %*tNEW VALUE",
*                 (2+offset), (10+offset), (22+offset), (45+offset));
*
         LA    R15,HEADER_O
         LA    R0,256
         LOGLIT1 '%*tOBS %*tVARIABLE %*tPREVIOUS VALUE %*tNEW VALUE'
         STM   R15,R1,PARMLIST
         L     R14,OUTHPARM+1*4
         LA    R1,45(R14)
         LA    R0,22(R14)
         LA    R15,10(R14)
         LA    R14,2(R14)
         STM   R14,R1,PARMLIST+3*4
         LA    R1,PARMLIST
         L     R15,&SAS_XPSSTR
         BALR  R14,R15
         ST    R15,I_O
*
*  length = 54 + offset;
*
         LA    R0,54
         A     R0,OUTHPARM+1*4
         ST    R0,LENGTH_O
*
*  if (SUMOPT) {
*
         L     R15,OUTHPARM
         LTR   R15,R15
         BZ    X0020
*
*     SAS_XPSSTR(header+(i+13),256-(i+13), "SUM");
*
         L     R1,I_O
         LA    R1,13(R1)
         LA    R14,HEADER_O(R1)
         LA    R15,256
         SR    R15,R1
         LA    R0,SUM_LIT
         STM   R14,R0,PARMLIST
         LA    R1,PARMLIST
         L     R15,&SAS_XPSSTR
         BALR  R14,R15
*
*     length += 17;
*
         L     R15,LENGTH_O
         LA    R0,17(R15)
```

```
*
*  SAS_XPHDR(1, header, length);
*
X0020    DS    0H
         LA    R14,1
         LA    R15,HEADER_O
         STM   R14,R0,PARMLIST
         L     R15,&SAS_XPHDR
         BALR  R14,R15
*
*  SAS_XPAGE();
*
         SLR   R1,R1
         L     R15,&SAS_XPAGE
         BALR  R14,R15
         L     R14,OUTHEAD#
         LM    R0,R7,OUTHEAD#+8
         BR    R14
         SPACE 3
MAIND    DSECT
OUTHEAD# DS    10F
OUTHPARM DS    8C
HEADER_O DS    256C
I_O      DS    F
LENGTH_O DS    F
MAIN     CSECT
         EJECT
*
PRNTOBS  DS    0H
*
*  /*----Format the numbers and print them to the print file ------*/
*  void    Prnt_obs(obser, old_obser, printopt, varinfo)
*  double  obser,old_obser;
*  struct SYSOPTS printopt;
*  struct UWINFOSTR varinfo;
*
         STM   R14,R7,PRNTOBS#
         MVC   PNTOPARM(24),0(R1)
*
*  SAS_XPS("%*t%*s", (11+printopt.offset), varinfo.namelen,
*          varinfo.name);
*
         LOGLIT1 '%*t%*s'
         L     R2,PNTOPARM+2*8+0*4
         L     R3,PNTOPARM+2*8+1*4
         L     R14,OFFSET-PRINTOPT(R2)
         LA    R14,11(R14)
         L     R15,VNAMELEN-VARINFO(R3)
         L     R0,VNAMEPTR-VARINFO(R3)
         STM   R14,R0,PARMLIST+4
         LA    R1,PARMLIST
         L     R15,&SAS_XPS
         BALR  R14,R15
*
*  SAS_XPS("%*t%14.*f", (22+printopt.offset), varinfo.nfcode,
*          old_obser);
*
         LOGLIT1 '%*t%14.*f'
         L     R14,OFFSET-PRINTOPT(R2)
         LA    R14,22(R14)
         L     R15,VNFCODE-VARINFO(R3)
         STM   R14,R15,PARMLIST+4
* skip 4 bytes to ensure doubleword alignment
         MVC   PARMLIST+16(8),PNTOPARM+0*8
         LA    R1,PARMLIST
         L     R15,&SAS_XPS
         BALR  R14,R15
*
*  SAS_XPS("%*t%14.*f", (40+printopt.offset), varinfo.nfcode, obser);
*
         L     R14,OFFSET-PRINTOPT(R2)
         LA    R14,40(R14)
         ST    R14,PARMLIST+4
* skip 4 bytes to ensure doubleword alignment
         MVC   PARMLIST+4*4(8),PNTOPARM+1*8
         LA    R1,PARMLIST
         L     R15,&SAS_XPS
         BALR  R14,R15
*
         L     R14,PRNTOBS#
         LM    R0,R7,PRNTOBS#+8
         BR    R14
         SPACE 3
MAIND    DSECT
PRNTOBS# DS    10F
PNTOPARM DS    0D,24C
MAIN     CSECT
         EJECT
```

(continued on next page)

Listing 11.3 (continued)

```
PRNTSUM  DS    0H
*  /*--------- Add the sum to the output if necessary -------------*/
*  void    Prnt_sum(sum, offset, varinfo)
*  double  sum;
*  int     offset;
*  struct UWINFOSTR *varinfo;
*
*  SAS_XPS("%*t%15.*f", (55+offset), varinfo.nfcode+2, sum);
*
         STM   R14,R7,PRNTSUM#
         MVC   PNTSPARM(16),0(R1)
         LOGLIT1 '%*t%15.*f'
         L     R14,PNTSPARM+1*8
         LA    R14,55(R14)
         L     R15,PNTSPARM+1*8+1*4
         L     R15,VNFCODE-VARINFO(R15)
         AL    R15,=F'2' (don't do la since R15 may have hi bit on)
         STM   R14,R15,PARMLIST+4
* skip 4 bytes to ensure doubleword alignment
         MVC   PARMLIST+16(8),PNTSPARM
         LA    R1,PARMLIST
         L     R15,&SAS_XPS
         BALR  R14,R15
         L     R14,PRNTSUM#
         LM    R0,R7,PRNTSUM#+8
         BR    R14
MAIND    DSECT
PRNTSUM# DS    10F
PNTSPARM DS    0D,16C
MAIN     CSECT
         LTORG
         EJECT
*
*-------------------------------------------------------------------*
* This is a required routine that accompanies MAIN. It is called by  *
* SAS_XEXIT processing. EXIT's responsibility is to return to the    *
* address that is immediately after the BALR R14,R15 that first      *
* brought us into MAIN. We find this out by picking up the original  *
* caller's savearea ptr from our SAVEAREA location. We can restore   *
* regs and return, based on its value.                               *
*-------------------------------------------------------------------*
*
         ENTRY EXIT
         USING EXIT,R15
EXIT     DS    0H
         L     R13,SAVEAREA+4-MAIND+NONRENT (orig caller's SA)
         LM    R14,R12,12(R13)         Restore regs to orig caller
         BR    R14                     Back to that caller
*
         DS    0D
NONRENT  DS    (MAINDL)C
SUM_LIT  DC    C'SUM',X'00'
*-----various data fields------------------------------------------*
MAIND    DSECT
SAVEAREA DS    18F                     our save area
         DS    F                       extra fullword used by C
DOUBLE   DS    D
PARMLIST DS    0D,50F                  parmblock
RC       DS    5F                      return code array
PID      DS    A                       poolid
FID      DS    A                       fileid
P        DS    A                       general pointer
I        DS    F                       general index
J        DS    F                       general index
PROCSTR  DS    0D,(PROCL)C             PROC structure area
NAMESTRC DS    0D,(NAMESTRL)C          NAMESTR structure area
INFILE   DS    A        /* pointer to the input data set.     */
OUTFILE  DS    A        /* pointer to the output data set.    */
PTRXV    DS    A        /* pointer for SAS_XVGET calls        */
OUTPTR   DS    A        /* pointer for SAS_XVPUT calls        */
*                       /* all the allocated memory comes from.*/
LIST     DS    A        /* the list of variable numbers.      */
NVAR     DS    H        /* the number of variables in the list.*/
X        DS    F        /* a counter variable.                */
OBS      DS    F        /* a counter of the # of observations. */
SUMOPT   DS    X        /* a boolean                          */
*                       /*      1= print sum in printed output */
*                       /*      0= do not print sum in output. */
PRINT    DS    X        /* a boolean                          */
*                       /*      1= output to a print file     */
*                       /*      0= Do not output to print file.*/
ALLMISS  DS    X        /* a boolean                          */
*                       /*   1= all the variables were missing */
*                       /*   an observation.                  */
*                       /*   0= there was at least one obs    */
*                       /*      that was not missing.         */
MULT     DS    D        /* the number to multiply by.         */
OBSERVAT DS    A        /* pointer to the memory that will    */
*                       /* hold the observations.             */
OLDOBSER DS    A        /* hold the old observations after    */
*                       /* multiplying                        */
```

```
SUM       DS    D        /* the sum of all the variables for one*/
*                        /* observation.                         */
PRINTSUM DS     A        /* hold the running sum for the        */
*                        /* variables in the observation        */
VARINFOPTR DS   A
PRINTOPTC DS    0F,(PRINTOPTL)C
SUMVAR    DS    (NAMESTRL)C
XBYPARM   DS    A
XVGETPRM DS     2F
XVPUTPRM DS     2F
XOADDPRM DS     2F
NAMPTR    DS    A
MAINDL    EQU   *-MAIND
PRINTOPT DSECT
CENTER    DS    F
DATE      DS    F
PLABEL    DS    F
OFFSET    DS    F
LINESIZE DS     F
PAGESIZE DS     F
PRINTOPTL EQU   *-PRINTOPT
VARINFO  DSECT
VNAMEPTR DS     A
VNAMELEN DS     F
VNFCODE  DS     F
VARINFOL EQU    *-VARINFO
         END
```

Part 4

Informats, Formats, Functions, and CALL Routines

Chapter 12 Understanding SAS® Informats, Formats, Functions, and CALL Routines

Introduction

This chapter describes how SAS informats, formats, functions, and CALL routines (IFFCs) interact with the SAS Supervisor and discusses grouping these user-written modules into packages. For instructions on how to write these modules, refer to the following chapters:

- Chapter 13, "Writing a SAS Informat"
- Chapter 14, "Writing a SAS Format"
- Chapter 15, "Writing a SAS Function or CALL Routine."

If you are writing your module in the FORTRAN, PL/I, or IBM 370 assembler languages, also refer to Chapter 16, "Sample IFFCs in the FORTRAN, PL/I, and IBM 370 Assembler Languages."

For information on compiling and linking modules, refer to the appendix for your operating system.

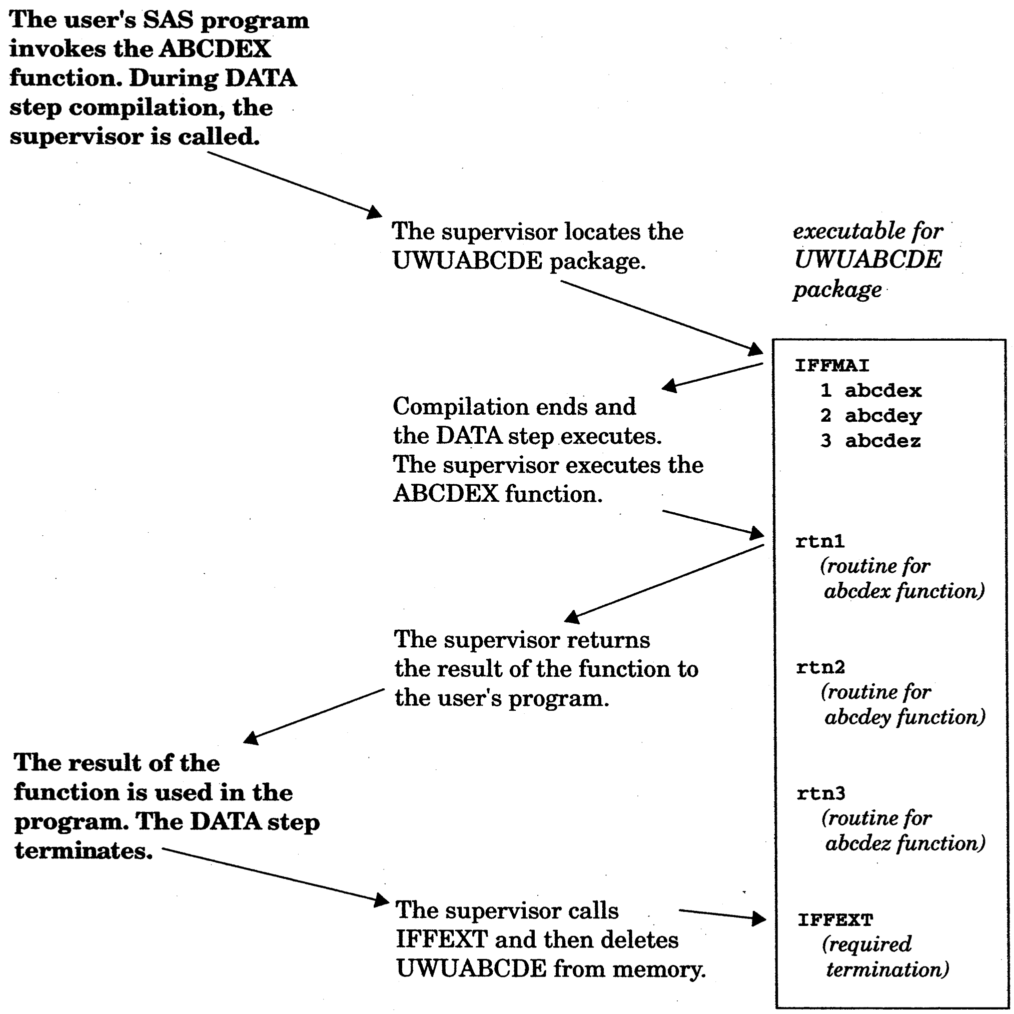
Grouping IFFCs into Packages

You can include one to ten user-written IFFCs in a single executable file, which is called an IFFC package. The term *IFFC package* is used throughout this chapter to refer to any number of IFFCs in a single file, even if you have only one. There are several rules you must follow when creating an IFFC package.

- All routines in a package must be of the same type. That is, the package must have only formats, or only informats, or only functions and CALL routines.
- The names of all routines in a package must start with the same five characters, excluding required prefixes such as $ (dollar sign) for character formats and informats and @ (at sign) for informats.
- The executable module for the IFFCs must be named UW*xchars*, where *x* is I for informats, F for formats, and U for functions or CALL routines, and *chars* are the first five characters that all IFFCs in the package have in common.

When you compile and link a package of IFFCs, the supervisor knows the package by the single name of the executable module (UW*xchars*). When the supervisor needs to call a single routine, it obtains the location of that specific routine from the IFFMAI routine in your package. For example, assume that the ABCDEX IFFC used in Figure 12.1 is a function. It is stored in an executable module named UWUABCDE. This figure illustrates how the supervisor communicates between the IFFC package and the user's program.

Figure 12.1
IFFCs and the Supervisor



Naming Conventions

When you compile and link an IFFC package, you must use a special prefix followed by the five-character identifier of the package for the name of the executable module. The prefix must be one of the following:

UWF indicates the package contains formats.

UWI indicates the package contains informats.

UWU indicates the package contains functions or CALL routines.

Thus, all IFFCs in a package have the same first five characters, and the name of the package must include those five characters after the prefix indicating the type of IFFC in the package.

After you have compiled and linked an IFFC package, you must set up the appropriate fileref before you invoke the SAS session that uses the IFFC. Refer to the appendix for your operating system for more detailed information.

For example, the executable module for the ABCDEX function in Figure 12.1 must be name UWUABCDE. The user sets up the SASLIB library to access the executable module for the ABCDE package. When the user invokes ABCDEX, the SAS Supervisor recognizes that it is not an Institute-supplied function after it tries to find it among the list of Institute-supplied IFFCs. Next, the supervisor looks for an executable module called UWUABCDE in the library of user-written modules, finds it, and then loads it.

IFFCs and the Supervisor

Once the supervisor determines the correct executable module for an IFFC, the supervisor loads the module and calls IFFMAI, a required routine in all IFFC packages, to find out the names of the IFFCs in the package. After the supervisor verifies that the IFFC exists, it then marks the executable module so it can be deleted from memory if the space is needed for compiling the DATA or PROC step that invoked the IFFC. When the execution phase begins, the supervisor reloads the executable module if necessary, and calls IFFMAI again. With this call, the supervisor asks for the locations of each IFFC routine to be called. As execution continues, the supervisor calls each requested IFFC as it is needed.

When the DATA or PROC step terminates, the supervisor calls another required routine in your IFFC package: IFFEXT. This routine performs any cleanup actions needed by your routine. When IFFEXT is completed, the supervisor deletes the executable module for your IFFC from memory.

Note: In some cases, you may want to use SAS_X or SAS_Z routines. If so, you must call the UWPRCC (for C or assembler), UWPRCF (for FORTRAN), or UWPRCP (for PL/I) routine to set up the interface between your IFFC module and the SAS/TOOLKIT environment. If your IFFC accesses external files, use the I/O routines appropriate for your language (for example, `fopen` for C).

Chapter 13 Writing a SAS® Informat

Introduction

This chapter describes how to write a SAS informat. You can include one to ten informats in the same C, FORTRAN, PL/I, or IBM 370 assembler program. This chapter also provides some brief information on informats and describes the sample informats used in this chapter. For more detailed information on informats, see the section "SAS Informats" in Chapter 3, "Components of the SAS Language," in *SAS Language: Reference, Version 6, First Edition.*

Character and Numeric Informats

An *informat* is an instruction that the SAS System uses to read data values into a variable. Informats can be either character or numeric. The type of informat determines how the data are stored after they are read. In the name of the informat you specify in your program, the first character in the name must be @ (at sign). Note that when you invoke the informat in a DATA or PROC step, you do not use the @ in the name. If the informat is for character data, the second character in the name of the informat must be $ (dollar sign). Unlike the @, the $ is used in the name when you invoke the informat.

This chapter uses two informats, one character and one numeric, to illustrate how to write a program to process informats. These informats are described in the following sections.

TESTINw.d

The TESTIN informat is a numeric informat with a width range of 1 to 16 and a default width of 8. The TESTIN informat reads a numeric value, shifts each number in the value to the next higher digit, and stores the transformed value. For example, if the input value is 123.45, the stored value is 234.56.

$TESTIC

The $TESTIC informat is a character informat with a width range of 1 to 16 and a default width of 8. The $TESTIC informat changes the input character string as shown in the following table.

Input String	Informatted Value
ABCDEFGH	PART 1
SECONDVL	PART 2
THIRDVL	PART 3
any other value	NOTFOUND

Naming Informats in an Executable Module

The name for an informat can have up to eight characters after the @ prefix. For character informats, the $ uses one of the eight characters, so only seven additional characters are permitted. If you want to include more than one informat in the same executable module, the first five characters after the @ and $ prefixes must be the same for all of the informats in the executable module. For example, the following informats can be in the same executable module:

@$CHANGC1
@$CHANGC2
@CHANGEN1
@CHANGEN2

This is useful when you want to create a number of related informats. If you need to use different names to make the name of the informat more meaningful, simply create a separate executable module for each informat.

Note: The SAS Supervisor looks at only the first five characters of the informat specified by the user to locate the executable module. Once the executable module is located, the supervisor can find the appropriate routine to process each informat. Note, also, that when you have informats that begin with the same first five characters, they *must* all be located in the same executable module.

Naming conventions for the executable modules are discussed in the appendix for your host operating system.

Required Prologue

The following sections describe the requirements for beginning a user-written SAS informat in the C language.

Accessing SAS/TOOLKIT Header Files

At the beginning of your program include the SAS/TOOLKIT header files. Use the following statement to access all the necessary definitions for writing programs in C:

```
#include <uwproc>
```

The `uwproc` include file contains many `#define` and `typedef` definitions, as well as other host-specific include files. After including `uwproc`, you can include any other header files or `#define` statements needed for your program.

Note: Be sure to use the `<uwproc>` format for the `#include` statement, not `"uwproc.h"`. Angle brackets are valid on all systems that run SAS/TOOLKIT software.

Communicating with the SAS Supervisor

When a user invokes an informat you have written, your program is called twice by the SAS Supervisor. The first time the supervisor calls your informat, you simply provide information about the informats that are stored in the executable file. The next time the supervisor calls the informat, it is executing the step that uses the informat, and it needs to know the address of the routine that processes the informat. To communicate these two types of information to the supervisor, the first C function in your program must be `iffmai`. The declaration of the `iffmai` routine is shown here:

```
ptr iffmai(request)
int *request;
{
statements for the iffmai function
}
```

The `iffmai` function checks the value of an integer set by the supervisor (the value pointed to by `request`) to see whether this is the first or second call to the informat. The `iffmai` routine then updates the value pointed to by `request` with the information needed by the supervisor.

The First Call by the Supervisor

The structure of the `iffmai` routine must include the following for the first time the supervisor calls your informat (that is, when `request` points to a value of 1):

- one call to the FMTDFS routine to identify the number of informats that are included in this executable module.
- one or more calls to the FMTDFN routine to describe the informats. Each call to FMTDFN provides the following information:
  - the number of the routine that processes the informat (for more detailed information, see "Writing the Routine to Process an Informat" later in this chapter)
  - the name of the informat
  - minimum width of the informat
  - maximum width of the informat
  - default width of the informat
  - minimum number of decimal places
  - maximum number of decimal places
  - default number of decimal places
  - justification (this is not applicable to informats; therefore, it should be set to 0 for all informats).
- one call to the FMTDFE routine to obtain the appropriate value to return to the supervisor.

The following illustrates how to code the first part of the `iffmai` routine. Note that both informats permit a width of 1 to 16, and both default to a width of 8. The numeric informat, TESTIN, permits from 0 to 10 decimal places and defaults to no decimals.

```
    /* first call from supervisor, identify executables */
if (*request == 1) {
    UWPRCC(0);
    FMTDFS(2);
    FMTDFN(1,"@TESTIN ",1,16,8,0,10,0,0);
    FMTDFN(2,"@$TESTIC",1,16,8,0, 0,0,0);
    return(FMTDFE());
    }
```

Note: If your informat needs to use any of the SAS_X or SAS_Z routines, you must include the following statement in the first part of the `iffmai` routine:

```
UWPRCC(0);
```

The UWPRCC routine is the SAS/TOOLKIT routine that sets up the interface between your C program and the SAS/TOOLKIT environment. For informats written in FORTRAN, call UWPRCF; for PL/I, call UWPRCP; for IBM 370 assembler, use the same routine as for C, UWPRCC.

The Second Call by the Supervisor

The second time the supervisor calls your informat (that is, when `request` points to a value of 2), the supervisor needs only the address of the routine that processes the informat. You provide this information simply by calling the FMTFNE routine and returning to the supervisor the value returned by FMTFNE.

```
      /* second call, provide address of routines by calling FMTFNE */
   else if (*request == 2) {
      return(FMTFNE());
      }
   }
```

Writing the Routine to Process an Informat

After the `iffmai` function, your C program must provide a function that processes each of the informats described in the first part of the `iffmai` routine.

Naming the Processing Routine

You must name the functions that process informats `rtn1` through `rtn10`. The number of the routine to process an informat corresponds to the value of the first argument to the FMTDFN routine. For example, the calls to FMTDFN for the two sample informats specify 1 and 2 for the first argument.

```
FMTDFN(1,"@TESTIN ", . . .);
FMTDFN(2,"@$TESTIC", . . .);
```

Thus, the TESTIN informat is processed by `rtn1`, and the $TESTIC informat is processed by `rtn2`. Note that you do not have to use all of the routines, `rtn1` through `rtn10`, but you must be sure that the number specified as the first argument to FMTDFN matches the number specified in the processing routine. In addition, the highest number used for one of the processing routines must match the number you specify in the call to FMTDFS. That is, if you define `rtn1`, `rtn3`, and `rtn5`, you must specify 5 in the call to FMTDFS.

Required Arguments for Each Routine

Each routine must have the arguments described in Table 13.1.

Table 13.1
Arguments for Informat Routines

Data Type	Variable	Use	Description
`ptr`	*from*	input	pointer to the input string.
`int`	*inw*	input	width of the input string.

(continued)

Table 13.1 (continued)

Data Type	Variable	Use	Description
`int`	*w*	input	width of the output string. This should always be the size of a double.
`int`	*d*	input	usually the number of decimal places. Use this value as a divisor on the numeric value before you return it, or if you are using SAS_XFXIN to informat a numeric value, specify this value for the number of decimal places. If you are writing a character informat, ignore this value. You must still declare it as an argument to your function, however.
`double` or `ptr`	**to* or *to*	output	pointer to the output value. For numeric informats, this points to the double-precision value of the number. If you are writing a character informat, this is a pointer to the returned string.

Processing a Numeric Informat

Once you have defined the function to process the informat, you can program the informatting as needed. If you want to use any of the SAS_X or SAS_Z routines, be sure that the first part of the the `iffmai` function includes a call to UWPRCC (or UWPRCF for FORTRAN, UWPRCP for PL/I).

The sample numeric format uses the SAS_ZMOVEI routine to create a working copy of the input string. It then calls SAS_ZSTRANS to change the digits in the working copy. Finally, the sample calls SAS_XFXIN to change the character string containing digits to a double precision floating-point value. The informatted value produced when the user invokes TESTIN is stored in the area pointed to by `to`, which can be accessed by the supervisor. The value returned by SAS_XFXIN indicates whether or not the call to SAS_XFXIN is successful. The `rtn1` routine passes the same return code back to the supervisor.

```
int rtn1(from,inw,w,d,to)
ptr from;
int inw,w,d;
double *to;
{
char temp[33];

 /*-----copy into work area-------------------------------------------*/
SAS_ZMOVEI(from,temp,inw);

 /*-----translate each digit to next higher digit---------------------*/
SAS_ZSTRANS(temp,inw,"1234567890","0123456789",10);

 /*-----input the new string into the returning number location-------*/
return(SAS_XFXIN(temp,inw,d,0,to));
}
```

Processing a Character Informat

Character informats differ from numeric informats by returning a pointer to a character string instead of returning a pointer to a double. Note that the last argument in the definition of `rtn2` has a type of `ptr`, which is a pointer to a character.

```
int rtn2(from,inw,w,d,to)
ptr from;
int inw,w,d;
ptr to;
{
statements for the rtn2 function
}
```

The $TESTIC informat is a simple table lookup informat. The following statements define the two tables of valid input values and the appropriate informatted value. Note that the second table has one more entry than the first. The last entry in the table of informatted values is used for any input value that doesn't match one of the values in the input table.

```
static char table[3][8] =      {{'A','B','C','D','E','F','G','H'},
                                {'S','E','C','O','N','D','V','L'},
                                {'T','H','I','R','D','V','L',' '}};

static char resp [4][8] =      {{'P','A','R','T',' ','1',' ',' '},
                                {'P','A','R','T',' ','2',' ',' '},
                                {'P','A','R','T',' ','3',' ',' '},
                                {'N','O','T','F','O','U','N','D'}};
```

The main action of the informat is to scan the table of input values looking for a match. The value in the informatted table is moved to the area pointed to by `to`, which is where the supervisor finds the informatted value. Note that this routine returns 0, not the return value from SAS_ZSTRMOV. The value returned by SAS_ZSTRMOV is the length of the string that is moved, which would be an inappropriate return code to pass to the supervisor.

```
int i;

   /* look for a match in the tables */
for (i=0;i<3 & SAS_ZSTRCOM(from,inw,table[i],8);i++);

   /* copy the proper table entry into place */
SAS_ZSTRMOV(resp [i],8,to,w);

return(0);
}
```

Required Termination Routine

After you have completed all of the routines to process informats in an informat package, you must include the IFFEXT routine for termination processing. In many cases, the IFFEXT routine you code in your program is as simple as the following:

```
void iffext() {}
```

If you have processing that you want to perform before your informat package is terminated, you can include that processing in this routine. For example, if you have allocated memory using a native language statement instead of one of the SAS_X memory allocation functions, be sure to free the allocated memory in the IFFEXT routine.

Note: If you plan to write many IFFC packages, and you do not intend to perform any termination processing, you may prefer to define the IFFEXT routine in a separate source module and simply link this module to every IFFC package you write.

Complete Example of an Informat Package

The following program illustrates the complete example explained in parts earlier in this chapter. Refer to Chapter 16, "Sample IFFCs in the FORTRAN, PL/I, and IBM 370 Assembler Languages," later in this book for illustrations of the same informats in other supported languages.

Listing 13.1
C Informat Example

```
/*---------------------------------------------------------------*/
/* NAME:          INFMTC                                         */
/* TYPE:          INFORMAT                                       */
/* LANGUAGE:      C                                              */
/* PURPOSE:       Example informat package to demonstrating writing   */
/*                SAS/Toolkit informats                          */
/* NOTES:         There is also a PL/I, FORTRAN, and IBM ASM/370 */
/*                version of this routine.                       */
/*                This source with be linked into the UWUTESTI   */
/*                executable. This is necessary since all entities */
/*                defined herein start with 'TESTI' (excluding @ and */
/*                '$').                                          */
/*---------------------------------------------------------------*/

/*-----the one necessary #include file---------------------------*/

#include <uwproc>

/*---------------------------------------------------------------*/
/* The iffmai routine is required. It is called by the SAS supervisor*/
/* at least once for request=1 and exactly once for request 2.   */
/* Request 1 is to obtain the names and attributes of the informats. */
/* Request 2 is to obtain the addresses of the routines to be called.*/
/*---------------------------------------------------------------*/

ptr iffmai(request)
int *request;
{

/*---------------------------------------------------------------*/
/* With request 1, we call certain routines. The first routine we  */
/* call is UWPINI. This routine is only necessary if you are going */
/* to use SAS_ interface routines in your code. We do indeed in this */
/* example, so the UWPINI call is necessary. Ensure a 0 is passed to */
/* it. The next call made is to FMTDFS. We pass this routine the  */
/* number of entities (IFFs) being defined. We then follow with as */
/* many FMTDFN calls as there are entities. With FMTDFN, we give the */
/* entity number which will match with the corresponding rtn      */
/* routine (e.g., entity 1 corresponds to rtn1). We also give the */
```

```
/* entity name (8-character padded string), the minimum/maximum/     */
/* default width, and the mininum/maximum/default number of decimals.*/
/* The last argument, the justification code, is not applicable to   */
/* informats and will ignored. The last call is to FMTDFE, which     */
/* signals the end of informat/format definition. Its return code is */
/* returned to the supervisor.                                        */
/* Note that FMTDFN is used both for formats and informats. Informats*/
/* are specified with a leading '@'. Note also that informats and    */
/* formats cannot be mixed in the same package.                      */
/*-------------------------------------------------------------------*/

if (*request == 1) {
    UWPRCC(0);
    FMTDFS(2);
    FMTDFN(1,"@TESTIN ",1,16,8,0,10,0,0);
    FMTDFN(2,"@$TESTIC",1,16,8,0, 0,0,0);
    return(FMTDFE());
    }

/*-------------------------------------------------------------------*/
/* With request 2, we will only need to call the FMTFNE routine,     */
/* which will pass to the supervisor the necessary info concerning   */
/* the addresses of the rtnx routines. We need to return the return  */
/* code of FMTFNE. Note that if you have any kind of initialization  */
/* code for some external process, this is the place to put it.      */
/*-------------------------------------------------------------------*/

else if (*request == 2) {
   return(FMTFNE());
   }
}

/*-------------------------------------------------------------------*/
/* rtn1 is the first informat  we are defining. Note that all infmts */
/* will use the same calling sequence: the first argument is the     */
/* pointer to the input string,the second is the input               */
/* width, the third is the width of the output data, fourth is the   */
/* number of decimals (always ignored for character), and fifth is   */
/* the pointer to the output field. This last pointer is a dblptr    */
/* for numeric informats, and a straight ptr for character informats.*/
/* This first informat routine, rtn1, corresponds to TESTIN. This in-*/
/* format will shift down each numeric character in the input string */
/* and then read in the new number. For example, '234' will become   */
/* '123' and be read in as 123. Note that the return code from the   */
/* SAS_XFXIN is returned to the supervisor. In general, any nonzero  */
/* return code means the informat was not successful.                */
/*-------------------------------------------------------------------*/

int rtn1(from,inw,w,d,to)
ptr from;
int inw,w,d;
double *to;
{
char temp[33];

/*-----copy into work area-------------------------------------------*/
SAS_ZMOVEI(from,temp,inw);

/*-----translate each digit to next lower digit----------------------*/
SAS_ZSTRANS(temp,inw,"1234567890","0123456789",10);

/*-----input the new string into the returning number location-------*/
return(SAS_XFXIN(temp,inw,d,0,to));
}

/*-------------------------------------------------------------------*/
/* rtn2 corresponds to the informat $TESTIC. This informat will      */
/* attempt a small table lookup. If ABCDEFGH is seen, it will return */
/* PART 1. If SECONDVL is seen, it will return PART 2. If THIRDVL    */
/* is seen, it will return PART 3. If none of these is seen, it will */
/* return NOT FOUND.                                                  */
/*-------------------------------------------------------------------*/

int rtn2(from,inw,w,d,to)
ptr from;
int inw,w,d;
ptr to;
{

/*-----input values--------------------------------------------------*/
static char table[3][8] =      {{'A','B','C','D','E','F','G','H'},
                                {'S','E','C','O','N','D','V','L'},
                                {'T','H','I','R','D','V','L',' '}};

/*-----output values-------------------------------------------------*/
static char resp [4][8] =      {{'P','A','R','T',' ','1',' ',' '},
                                {'P','A','R','T',' ','2',' ',' '},
                                {'P','A','R','T',' ','3',' ',' '},
                                {'N','O','T','F','O','U','N','D'}};
```

(continued on next page)

Listing 13.1
(continued)

```
int i;

/*---------------------------------------------------------------*/
/* Note here the use of SAS_ZSTRCOM and SAS_ZSTRMOV. The SAS_ZSTRCOM */
/* routine will compare two fields with differing lengths, by assum- */
/* ing blank padding for the shorter. It returns 0 if a match, and   */
/* nonzero otherwise. SAS_ZSTRMOV moves from one location to another */
/* using the specified widths. If the destination length is longer, */
/* it pads with blanks. If shorter, it truncates.                    */
/*---------------------------------------------------------------*/

/*-----look for a match in the tables--------------------------------*/
for (i=0;i<3 & SAS_ZSTRCOM(from,inw,table[i],8);i++);

/*-----copy the proper table entry into place------------------------*/
SAS_ZSTRMOV(resp [i],8,to,w);

return(0);
}
```

Chapter 14 Writing a SAS® Format

Introduction

This chapter describes how to write a SAS format. You can include one to ten formats in the same C, FORTRAN, PL/I, or IBM 370 assembler program. This chapter also provides some brief information on formats and describes the sample formats used in this chapter. For more detailed information on formats, see the section "SAS Formats" in Chapter 3, "Components of the SAS Language," in *SAS Language: Reference, Version 6, First Edition.*

Character and Numeric Formats

A *format* is an instruction that the SAS System uses to write data values. Formats can be either character or numeric. The type of format depends on how the data that you want to write are stored. Formats always produce character strings.

If the format is for character data, the first character in the name of the format must be $ (dollar sign).

This chapter uses two formats, one character and one numeric, to illustrate how to write a program to process formats. These formats are described in the following sections.

$ONEOFF

The $ONEOFF format is a character format with a width range of 1 to 200 and a default width of 1. The $ONEOFF format changes each character in a character string to the next character in the host operating system's collating sequence. That is, A is changed to B and so on.

ONEOFXw.d

The ONEOFX format is a numeric format with a width range of 1 to 32 and a default width of 1. ONEOFX formats numeric values in standard numeric representation and then changes leading blanks to # characters.

Naming Formats in an Executable Module

The name for a format can have up to eight characters. For character formats, the $ uses one of the eight characters, so only seven additional characters are permitted. If you want to include more than one format in the same executable module, the first five characters after the $ prefix must be the same for all of the formats in the executable module. For example, the following formats can be in the same executable module:

$CHANGC1
$CHANGC2
CHANGEN1
CHANGEN2

This is useful when you want to create a number of related formats. If you need to use different names to make the name of the format more meaningful, simply create a separate executable module for each format.

Note: The SAS Supervisor looks at only the first five characters of the format specified by the user to locate the executable module. Once the executable module is located, the supervisor can find the appropriate routine to process each format. Note, also, that when you have formats that begin with the same first five characters, they *must* all be located in the same executable module.

Naming conventions for the executable modules are discussed in the appendix for your host operating system.

Required Prologue

The following sections describe the requirements for beginning a user-written SAS format in the C language.

Accessing SAS/TOOLKIT Header Files

At the beginning of your program, include the SAS/TOOLKIT header files. Use the following statement to access all the necessary definitions for writing programs in C:

```
#include <uwproc>
```

The `uwproc` include file contains many `#define` and `typedef` definitions and also includes other host-specific include files. After including `uwproc`, you can include any other header files or `#define` statements needed for your program.

Note: Be sure to use the `<uwproc>` format for the `#include` statement, not `"uwproc.h"`. Angle brackets are valid on all systems that run SAS/TOOLKIT software.

Communicating with the SAS Supervisor

When a user invokes a format you have written, your program is called twice by the SAS Supervisor. The first time the supervisor calls your format, you simply provide information about the formats that are stored in the executable file. The next time the supervisor calls the format, it is executing the step that uses the format, and it needs to know the address of the routine that processes the format. To communicate these two types of information to the supervisor, the first C function in your program must be `iffmai`. The declaration of the `iffmai` routine is shown here:

```
ptr iffmai(request)
int *request;
{
```

The `iffmai` function checks the value of an integer set by the supervisor (the value pointed to by `request`) to see whether or not this is the first or second call to the format. The `iffmai` routine then updates the value pointed to by `request` with the information needed by the supervisor.

The First Call by the Supervisor

The structure of `iffmai` must include the following for the first time the supervisor calls your format:

- one call to the FMTDFS routine to identify the number of formats that are included in this executable module.
- one or more calls to the FMTDFN routine to describe the formats. Each call to FMTDFN provides the following information:
  - the number of the routine that processes the format (for more detailed information, see "Writing the Routine to Process a Format" later in this chapter)
  - the name of the format
  - minimum width of the format
  - maximum width of the format
  - default width of the format
  - minimum number of decimal places
  - maximum number of decimal places

  - default number of decimal places
  - justification (0 for left justification; 1 for right justification).

- one call to the FMTDFE routine to obtain the appropriate value to return to the supervisor.

The following illustrates how to code the first part of the `iffmai` routine for the two sample formats. Note that the character format, $ONEOFF, permits a width of 1 to 200 and defaults to 1. The numeric format, ONEOFX, permits a width of 1 to 32 and defaults to 1; it does not permit decimal places. In addition, both formats left-justify the formatted value.

```
   /* first call from supervisor, identify executables */
if (*request == 1) {
   UWPRCC(0);
   FMTDFS(2);
   FMTDFN(1,"$ONEOFF ",1,200,1,0,0,0,0);
   FMTDFN(2,"ONEOFX  ",1,32,1,0,0,0,0);
   return(FMTDFE());
   }
```

Note: If your format needs to use any of the SAS_X or SAS_Z routines, you must include the following statement in the first part of the `iffmai` function.

```
UWPRCC(0);
```

The UWPRCC routine is the SAS/TOOLKIT routine that sets up the interface between your C program and the SAS/TOOLKIT environment. For formats written in FORTRAN, call UWPRCF; for PL/I, call UWPRCP; for IBM 370 assembler, use the same routine as for C, UWPRCC.

The Second Call by the Supervisor

The second time the supervisor calls your format, it needs only the address of the routine that processes the format. You provide this information simply by calling the FMTFNE routine and returning to the supervisor the value returned by FMTFNE.

```
   /* second call, provide address of routines by calling FMTFNE */
else if (*request == 2) {
   return(FMTFNE());
   }
}
```

Writing the Routine to Process a Format

After the `iffmai` function, your C program must provide a function that processes each of the formats described in the first part of `iffmai`. The following sections describe how to name these functions and what arguments are required, and then discuss the processing in the sample formats.

Naming the Processing Routine

You must name the functions that process formats `rtn1` through `rtn10`. The number of the routine to process a format corresponds to the value of the first argument to the FMTDFN routine. For example, the calls to FMTDFN for the two sample formats specify 1 and 2 for the first argument.

```
FMTDFN(1,"$ONEOFF ", . . .);
FMTDFN(2,"ONEOFX  ", . . .);
```

Thus, the $ONEOFF format is processed by `rtn1`, and the ONEOFX format is processed by `rtn2`. Note that you do not have to use all of the routines `rtn1` through `rtn10`, but you must be sure that the number specified as the first argument to FMTDFN matches the number specified in the processing routine. In addition, the highest number used for one of the processing routines must match the number you specify in the call to FMTDFS. That is, if you define `rtn1`, `rtn3`, and `rtn5`, you must specify 5 in the call to FMTDFS.

Required Arguments for Each Routine

Each routine must have the arguments described in Table 14.1.

Table 14.1 *Arguments for Format Routines*

Data Type	Variable	Use	Description
`char` or `double`	**from*	input	pointer to the data to format. For numeric formats, this points to the double-precision value of the number. If you are writing a character format, this is a pointer to the string to format.
`int`	*inw*	input	width of the data to format. For numeric formats, this should always be the size of a double.
`int`	*w*	input	width of the formatted output.
`int`	*d*	input	number of decimal places. If you call SAS_XFXPN, use this value to specify the number of decimal places in the call. If you are writing a character format, ignore this value. You must still declare it as an argument to your function, however.
`char`	**to*	output	pointer to the formatted value. This is the value the supervisor accesses.

Processing a Character Format

Once you have defined the function to process the format, you can program the formatting as needed. If you want to use any of the SAS_X or SAS_Z routines, be sure that the first part of the `iffmai` function includes a call to UWPRCC (or UWPRCF for FORTRAN, UWPRCP for PL/I).

This sample illustrates a common use of SAS/TOOLKIT software for writing formats (instead of using the FORMAT procedure). Instead of looking up a conversion value in tables, like the VALUE statements of PROC FORMAT create, this format uses an algorithm to change the values. In this case, each character is changed to the next higher letter in the collating sequence on the host system by the first `for` statement. The next `for` statement sets the remaining bytes of the formatted value (pointed to by `to`) to blanks.

```
int rtn1(from,inw,w,d,to)
char *from;
int inw;
int w;
int d;
char *to;
{

int i,j;

j = MIN(inw,w);
for (i=0;i<j;i++) {
    to[i] = from[i] + 1;
    }

for (;i<w;) {
    to[i++] = ' ';
    }

return(0);
}
```

Processing a Numeric Format

Like character formats, the processing for numeric formats varies depending upon your needs. The primary difference between character and numeric formats is that the `from` argument points to a double precision floating-point value, not a character string.

In this format, the routine first formats the number pointed to by `from` by calling SAS_XFXPN. The formatted value is stored in a temporary character string. The format then uses SAS_ZSTRANC to translate the leading blanks to #, and uses SAS_ZMOVEI to move the final result to the area pointed to by `to`. The supervisor can access the formatted value at this location. The routine returns 0 to indicate it has successfully completed.

```
int rtn2(from,inw,w,d,to)
double *from;
int inw;
int w;
int d;
char *to;
```

```
{
char temp[32];

SAS_XFXPN(*from,32,d,0,temp);

SAS_ZSTRANC(temp,32,'#',' ');

SAS_ZMOVEI(temp+32-w,to,w);
return(0);
}
```

Required Termination Routine

After you have completed all of the routines to process formats in a format package, you must include the IFFEXT routine for termination processing. In many cases, the IFFEXT routine you code in your program is as simple as the following:

```
void iffext() {}
```

If you have processing that you want to perform before your format package is terminated, you can include that processing in this routine. For example, if you have allocated memory using a native language statement instead of one of the SAS_X memory allocation functions, be sure to free the allocated memory in the IFFEXT routine.

Note: If you plan to write many IFFC packages, and you do not intend to perform any termination processing, you may prefer to define the IFFEXT routine in a separate source module and simply link this module to every IFFC package you write.

Complete Example of a Format Package

The following program illustrates the complete example explained in parts earlier in this chapter. See Chapter 16, "Sample IFFCs in the FORTRAN, PL/I, and IBM 370 Assembler Languages," later in this book for illustrations of the same formats in other supported languages.

Listing 14.1
C Format Example

```
/*-----------------------------------------------------------------*/
/* NAME:         FMTC                                              */
/* TYPE:         FORMAT                                            */
/* LANGUAGE:     C                                                 */
/* PURPOSE:      Example format package to demonstrate how to write */
/*               SAS/Toolkit formats                               */
/* NOTES:        There is also a PL/I, FORTRAN, and IBM ASM/370    */
/*               version of this routine.                          */
/*-----------------------------------------------------------------*/

/*-----the one necessary #include file-----------------------------*/

#include <uwproc>

/*-----------------------------------------------------------------*/
/* The iffmai routine is required. It is called by the SAS supervisor*/
/* at least once for request=1 and exactly once for request 2.     */
```

(continued on next page)

Listing 14.1
(continued)

```
/* Request 1 is to obtain the names and attributes of the formats.   */
/* Request 2 is to obtain the addresses of the routines to be called.*/
/*-------------------------------------------------------------------*/

ptr iffmai(request)
int *request;
{

/*-------------------------------------------------------------------*/
/* With request 1, we call certain routines. The first routine we    */
/* call is UWPINI. This routine is only necessary if you are going   */
/* to use SAS_ interface routines in your code. We do indeed in this */
/* example, so the UWPINI call is necessary. Ensure a 0 is passed to */
/* it. The next call made is to FMTDFS. We pass this routine the     */
/* number of entities (IFFs) being defined. We then follow with as   */
/* many FMTDFN calls as there are entities. With FMTDFN, we give the */
/* entity number which will match with the corresponding rtn         */
/* routine (e.g., entity 1 corresponds to rtn1). We also give the    */
/* entity name (8-character padded string), the minimum/maximum/     */
/* default width, and the mininum/maximum/default number of decimals,*/
/* and the justification code (0=left 1=right). The last call made   */
/* is to FMTDFE, which signals the end of informat/format definition.*/
/* Its return code is returned to the supervisor.                    */
/*-------------------------------------------------------------------*/

if (*request == 1) {
   UWPRCC(0);
   FMTDFS(2);
   FMTDFN(1,"$ONEOFF ",1,200,1,0,0,0,0);
   FMTDFN(2,"ONEOFX  ",1,32,1,0,0,0,0);
   return(FMTDFE());
   }

/*-------------------------------------------------------------------*/
/* With request 2, we will only need to call the FMTFNE routine,     */
/* which will pass to the supervisor the necessary info concerning   */
/* the addresses of the rtnx routines. We need to return the return  */
/* code of FMTFNE. Note that if you have any kind of initialization  */
/* code for some external process, this is the place to put it.      */
/*-------------------------------------------------------------------*/

else if (*request == 2) {
   return(FMTFNE());
   }
}

/*-------------------------------------------------------------------*/
/* rtn1 is the first format    we are defining. Note that all formats*/
/* will use the same calling sequence: the first argument is the     */
/* pointer to the input data (note that this is a char * for char    */
/* formats, and a dblptr for numerics), the second is the input      */
/* width, the third is the width of the output data, fourth is the   */
/* number of decimals (always ignored for character), and fifth is   */
/* the pointer to the output field.                                  */
/* This first format   routine, rtn1, corresponds to $ONEOFF. This
/* format will shift each character in the input string to the next  */
/* largest character. For example, 'ABC' would be formatted as       */
/* 'BCD'. It pads with blanks if necessary.                          */
/*-------------------------------------------------------------------*/

int rtn1(from,inw,w,d,to)
char *from;                        /* I - ptr to data to format     */
int inw;                           /* I - width of input data       */
int w;                             /* I - width of output data      */
int d;                             /* I - no. of decimals (always 0)*/
char *to;                          /* I - ptr to output data        */
{

/* RETURNS:   always 0                                              */
/* END                                                              */
/*------------------------------------------------------------------*/

int i,j;

/*-----shift each input byte, up to w bytes--------------------------*/
j = MIN(inw,w);
for (i=0;i<j;i++) {
    to[i] = from[i] + 1;
    }

/*-----pad the remainder with blanks---------------------------------*/
for (;i<w;) {
    to[i++] = ' ';
    }

/*-----normal return code--------------------------------------------*/
return(0);
}

/*------------------------------------------------------------------*/
/* rtn2 is the next  format we are defining. This format routine    */
/* corresponds to ONEOFX. This format will first format out the     */
```

```
/* input number normally (using the SAS_XFXPN standard formatting    */
/* routine). Then, it converts all leading blanks to #.              */
/*-------------------------------------------------------------------*/

int rtn2(from,inw,w,d,to)
double *from;                      /* I - ptr to data to format     */
int inw;                           /* I - width of input data       */
int w;                             /* I - width of output data      */
int d;                             /* I - no. of decimals (always 0)*/
char *to;                          /* I - ptr to output data        */
{
char temp[32];

/*-----format normally---------------------------------------------*/
SAS_XFXPN(*from,32,d,0,temp);

/*-----translate blanks to #---------------------------------------*/
SAS_ZSTRANC(temp,32,'#',' ');

/*-----copy into target--------------------------------------------*/
SAS_ZMOVEI(temp+32-w,to,w);
return(0);
}
```

Chapter 15 Writing a SAS® Function or CALL Routine

Introduction

This chapter describes how to write a SAS function or CALL routine. You can include one to ten functions or CALL routines in the same C, FORTRAN, PL/I, or IBM 370 assembler program. This chapter also provides some brief information on functions and CALL routines and describes the samples used in this chapter. For more detailed information on functions and CALL routines, see the sections "SAS Functions" and "SAS CALL Routines" in Chapter 3, "Components of the SAS Language," in *SAS Language: Reference, Version 6, First Edition.*

Functions and CALL Routines

A SAS *function* is a builtin expression that returns a value resulting from zero or more arguments. A SAS *CALL routine* is a builtin expression that can change the value of any of the arguments passed to it, but does not return a value.

This chapter uses several functions and a CALL routine to illustrate how to write a program to process functions and CALL routines. These functions and CALL routines are described in the following sections.

TESTFADD and TESTFSUB Functions

The TESTFADD function has the following syntax:

z=TESTFADD(*x*,*y*);

The TESTFSUB function has the following syntax:

z=TESTFSUB(*x*,*y*);

where *x*, *y*, and *z* are all numeric values. The TESTFADD function adds the values of *x* and *y* and returns the sum in *z*. The TESTFSUB function subtracts the values. For both functions, if either *x* or *y* is missing, *z* is set to missing. Both of these functions are included in the complete example at the end of the chapter, but only the TESTFADD function is discussed in the illustrations in the chapter.

TESTFCHR Function

The TESTFCHR function has the following syntax:

return-string=TESTFCHR(*input-string*,*index*);

where *return-string* and *input-string* are character values, and *index* is a numeric value. This function changes dashes in the input string to another character depending on the value of *index*. The following table shows the new character value used for each value of *index*.

Value of *index*	New Character
1	)
2	|
3	@
4	#
5	$
6	%
7	¬
8	&
9	*
10	(

TESTFVC Function

The TESTFVC function has the following syntax:

return-string=TESTFVC(*char-1*<, . . . *char-10*>);

where *char-1* through *char-10* are character strings that are concatenated to form the *return-string*. This function illustrates how to handle a varying number of arguments.

TESTFCAL CALL Routine

The TESTFCAL CALL routine has the following syntax:

CALL TESTFCAL(*index-variable, char-variable*);

where *index-variable* must be a numeric variable set to a number from 1 to 5, and *char-variable* is any character variable. The TESTFCAL function changes both variables. The *index-variable* is always set to 99, and the *char-variable* is changed to the values listed in the following table based on the incoming value of *index-variable*.

Value of *index-variable*	New Value of *char-variable*
1	A
2	B
3	C
4	D
5	E

The following example illustrates the use of the TESTFCAL function:

```
x=2;
CALL TESTFCAL(x,y);
```

The result of these statements is X=99 and Y='B'.

Naming Functions or CALL Routines in an Executable Module

The name for a function or CALL routine can have up to eight characters. If you want to include more than one function or CALL routine in the same executable module, the first five characters must be the same for all of the functions or CALL routines in the module. For example, the following functions or CALL routines can be in the same executable module:

CHANGC1
CHANGC2
CHANGEN1
CHANGEN2.

This is useful when you want to create a number of related functions or CALL routines. If you need to use different names to make the name of the function or CALL routine more meaningful, simply create a separate executable module for each function or CALL routine.

Note: The SAS Supervisor looks at only the first five characters of the function or CALL routine specified by the user to locate the executable module. Once the executable module is located, the supervisor can find the appropriate routine to process each function or CALL routine. Note that this also means that when you have functions or CALL routines that begin with the same first five characters, they *must* all be located in the same executable module.

Naming conventions for the executable modules are discussed in the appendix for your host operating system.

Required Prologue

The following sections describe the requirements for beginning a user-written SAS function or CALL routine in the C language.

Accessing SAS/TOOLKIT Header Files

At the beginning of your program, include the SAS/TOOLKIT header files. Use this statement to access all the necessary definitions for writing programs in C:

```
#include <uwproc>
```

The `uwproc` include file contains many `#define` and `typedef` definitions, and also includes other host-specific include files. After including `uwproc`, you can include any other header files or `#define` statements needed for your program.

Note: Be sure to use the `<uwproc>` format for the `#include` statement, not `"uwproc.h"`. Angle brackets are valid on all systems that run SAS/TOOLKIT software.

Communicating with the SAS Supervisor

When a user invokes a function or CALL routine you have written, your program is called twice by the SAS Supervisor. The first time the supervisor calls your function or CALL routine, you simply provide information about the functions or CALL routines that are stored in the executable file. The next time the supervisor calls the function or CALL routine, it is executing the step that uses it, and it needs to know the address of the routine that processes the function or CALL routine. To communicate these two types of information to the supervisor, the first C function in your program must be `iffmai`. Begin `iffmai` as follows:

```
ptr iffmai(request)
int *request;
{
```

The `iffmai` function checks the value of an integer set by the supervisor (the value pointed to by `request`) to see whether this is the first or second call to the function or CALL routine. The `iffmai` routine then updates the value pointed to by `request` with the information needed by the supervisor.

The First Call by the Supervisor

The structure of `iffmai` must include the following for the first time the supervisor calls your function or CALL routine:

- one call to the FNCDFS routine to identify the number of functions or CALL routines that are included in this executable module.
- one or more calls to the FNCDFN routine to describe the functions or CALL routines. Each call to FNCDFN provides the following information:
  - the number of the routine that processes the function or CALL routine. (For more detailed information, see "Writing the Processing Routine" later in this chapter.)
  - the name of the function or CALL routine.
  - minimum number of arguments required when the user executes the function or CALL routine. This can be 0.
  - maximum number of arguments permitted when the user executes the function or CALL routine.
  - type of value returned (0 for no return value (a CALL routine), 1 for a numeric value, 2 for a character value).
- one call to FNCDFA for each of the first ten arguments in each routine. That is, call FNCDFA for the first ten arguments or, if the routine has fewer than ten, the maximum number of arguments for the routine. For each call to FNCDFA, provide the following information:
  - the number of the function using the argument
  - the number of the argument
  - the type of the argument (1 for numeric, 2 for character, 3 for either character or numeric).
- one call to the FNCDFE routine to obtain the appropriate value to return to the supervisor.

The following illustrates how to code the first part of the `iffmai` routine for the sample functions and CALL routines. In this example, the complete information for each routine is provided in a group. That is, the call to FNCDFN is followed immediately by the calls to FNCDFA to define the arguments for the routine. If you prefer, you can group all the calls to FNCDFN first, and follow them with all the calls to FNCDFA. The calls shown below define the routines as follows:

TESTFADD
: is processed by `rtn1`, has both a minimum and a maximum of 2 arguments, and returns a numeric value. Both arguments are numeric.

TESTFSUB
: is processed by `rtn2`, has both a minimum and a maximum of 2 arguments, and returns a numeric value. Both arguments are numeric.

TESTFCHR — is processed by `rtn3`, has both a minimum and a maximum of 2 arguments, and returns a character value. The first argument is character; the second is numeric.

TESTFVC — is processed by `rtn4`, has a minimum of 1 argument and a maximum of 10 arguments, and returns a character value. All arguments are character.

TESTFCAL — is processed by `rtn5`, has both a minimum and a maximum of 2 arguments, and returns nothing, so it is a CALL routine. The first argument is numeric; the second is character.

These routines are defined in the following code:

```
   /* first call from supervisor, identify executables */
if (*request == 1) {
    UWPRCC(0);
    FNCDFS(5);

    FNCDFN(1,"TESTFADD",2,2,1);
    FNCDFA(1,1,1);
    FNCDFA(1,2,1);

    FNCDFN(2,"TESTFSUB",2,2,1);
    FNCDFA(2,1,1);
    FNCDFA(2,2,1);

    FNCDFN(3,"TESTFCHR",2,2,2);
    FNCDFA(3,1,2);
    FNCDFA(3,2,1);

    FNCDFN(4,"TESTFVC ",1,10,2);
    FNCDFA(4,1,2);
    FNCDFA(4,2,2);
    FNCDFA(4,3,2);
    FNCDFA(4,4,2);
    FNCDFA(4,5,2);
    FNCDFA(4,6,2);
    FNCDFA(4,7,2);
    FNCDFA(4,8,2);
    FNCDFA(4,9,2);
    FNCDFA(4,10,2);

    FNCDFN(5,"TESTFCAL",2,2,0);
    FNCDFA(5,1,1);
    FNCDFA(5,2,2);

    return(FNCDFE());
    }
```

Note: If your function or CALL routine needs to use any of the SAS_X or SAS_Z routines, you must include the following statement in the first part of the `iffmai` routine.

```
UWPRCC(0);
```

The UWPRCC routine is the SAS/TOOLKIT routine that sets up the interface between your C program and the SAS/TOOLKIT environment. For functions or CALL routines written in FORTRAN, call UWPRCF; for PL/I, call UWPRCP; for IBM 370 assembler, use the same routine as for C: UWPRCC.

The Second Call by the Supervisor

The second time the supervisor calls your function or CALL routine, it needs only the address of the routine that processes the function or CALL routine. You provide this information simply by calling the FNCFNE routine, and returning to the supervisor the value returned by FNCFNE.

```
   /* second call, provide address of routines by calling FNCFNE */
else if (*request == 2) {
   return(FNCFNE());
   }
}
```

Writing the Processing Routine

After the `iffmai` function, your C program must provide a function that processes each of the functions or CALL routines described in the first part of the `iffmai` routine. The following sections describe how to name these functions and what arguments are required, and then discuss the processing in the sample functions and CALL routines.

Naming the Processing Routine

You must name the functions that process functions or CALL routines `rtn1` through `rtn10`. The number of the routine to process a function or CALL routine corresponds to the value of the first argument to the FNCDFN routine. For example, the calls to FNCDFN for the sample functions and CALL routine specify 1 through 5 for the first argument.

```
FNCDFN(1,"TESTFADD", . . .);
FNCDFN(2,"TESTFSUB", . . .);
FNCDFN(3,"TESTFCHR", . . .);
FNCDFN(4,"TESTFVC ", . . .);
FNCDFN(5,"TESTFCAL", . . .);
```

These functions and CALL routine are processed by `rtn1` through `rtn5`, respectively. Note that you do not have to use all of the routines `rtn1` through `rtn10`, but you must be sure that the number specified as the first argument to FNCDFN matches the number specified in the processing routine. In addition, the highest number used for one of the processing routines must match the number you specify in the call to FNCDFS. That is, if you define `rtn1`, `rtn3`, and `rtn5`, you must specify 5 in the call to FNCDFS.

Accessing the Arguments for a Routine

Functions and CALL routines can have numeric or character arguments, depending on the specifications for the function. When a function or CALL routine has a numeric argument, the `rtn`*n* routine has one corresponding argument. However, when the function or CALL routine has a character argument, the `rtn`*n* routine has three arguments that correspond to the single character argument. These three arguments are as follows:

- the address of a maximum length for the argument
- the address of the current length
- the address of the character data.

See "Processing Character Arguments" later in this chapter for more details on how to use this information.

When you define a function, the last argument (or triple of arguments for character) is the returning value. The function itself returns a return code to indicate the level of success in the function.

Functions and CALL routines can have a fixed number of arguments, or a varying number, also depending on specification. When the call to FNCDFN has the same value for the minimum and maximum number of arguments, the function or CALL routine has a fixed number of arguments. If you have a fixed number of arguments, the maximum number of values that you can pass to the `rtn`*n* routine is 100. This limit is not exactly the same as the number of arguments permitted on the function or CALL routine since each character argument in a function or CALL routine is actually three arguments in the corresponding `rtn`*n* routine.

If you have a varying number of arguments (the minimum and maximum values in the call to FNCDFN are different), you can permit any number of arguments in the function or CALL routine. See "Processing Varying Numbers of Arguments" later in this chapter for more information.

Processing Numeric Arguments

The following example illustrates how to declare the arguments for a simple numeric function that processes a fixed number of arguments. In this example, the function passes two numeric arguments and returns a numeric value pointed to by `arg3`. Note that the routine uses the SAS_ZMISS routine to check for missing values before performing any arithmetic on the values.

```
int rtn1(arg1,arg2,arg3)
double *arg1,*arg2,*arg3;
```

```
{
if (SAS_ZMISS(*arg1) || SAS_ZMISS(*arg2))
   return(F_EMISS);
*arg3 = *arg1 + *arg2;
return(F_OK);
}
```

When the routine has a fixed number of arguments, you can access the values of the arguments by using the arguments passed to the function. In C, you must dereference the addresses defined in the function's definition to get to the value. In this case, to access the numeric values, use the dereferencing operator (*). If your function permits a varying number of arguments, see the section "Accessing Varying Numbers of Arguments" later in this chapter.

Each numeric argument is actually passed as an address of a double-precision floating-point number. In PL/I and FORTRAN, since these languages are call-by-address languages, you can code the attribute of the argument as a double-precision number. In the C language, you must indicate that the argument is the pointer to a double. IBM 370 assembler language is treated as a call-by-value language, so it functions like C. Note that you can update the value of any numeric argument, since you have the pointer to its location. However, for SAS functions (those routines that return a single value), only the last argument's value is really updated by the supervisor. For a CALL routine, any numeric argument changed by the routine is updated by the supervisor.

Processing Character Arguments

Each character argument for a function or CALL routine is passed to the `rtnn` routine as three values that are all addresses. Note that when you define the arguments using FNCDFN and FNCDFA, you indicate only one argument for each character argument. When you create the routine that processes the function, however, you define three arguments for each character argument. The three values for each character argument are as follows:

- □ The first value is the address of a maximum length. Do not update this value in your routine. The maximum length is the maximum number of bytes that can be changed by the function.
- □ The second value is the address of the current length. This can be updated to reflect what the function does to the character data. However, this length must not exceed the maximum length.
- □ The third and last argument is the address of the character data. The data pointed to by this address can be updated up to the number of bytes indicated by the first argument of the triple.

As with numeric routines, if you have a fixed number of arguments, you can directly access the three values passed for each character argument by using the arguments defined in your processing routine. In this example, the first argument passed to the routine is a character argument; the second is a numeric argument; and the routine returns a character value. Each of the character arguments has three values defined in the processing routine, `m1`, `c1`, `p1` and `mr`, `cr`, `pr`. The `value` argument is numeric.

```
int rtn3(m1,c1,p1,value,mr,cr,pr)
short *m1,*c1;
ptr *p1;
double *value;
short *mr,*cr;
ptr *pr;
{
int j;
ptr p,q;
char i;

static char delims[10] = {')','|','@','#','$','%','¬','&','*','('};

i = delims[((int)*value)-1];

*cr = *c1;

p = *p1;
q = *pr;

for (j = *c1;j>0;j--,p++,q++) {
    if ((*q = *p) == '-')
       *q = i;
    }

return(F_OK);
}
```

Note that this example sets the length of the returning string to the maximum length of the input string. The input string is copied to the output area, and dashes are changed to the delimiter indicated by the `value` index.

Processing Arguments That Can Be Numeric or Character

If you specify that one of the arguments in a function or CALL routine can be either a numeric or a character value (by using 3 for the `type` value in the call to FNCDFA), you must use the FNCARG routine to retrieve the value of that argument *and all subsequent arguments*. The process for using FNCARG is described in more detail in the next section, but keep in mind that you must use FNCARG even when you have a fixed number of arguments if any of the arguments can be either numeric or character.

Accessing Varying Numbers of Arguments

Some functions can be called with a varying number of arguments (like the Institute-supplied SUM function). You should code your function to expect the minimum number of arguments given by FNCDFN. In this example, the minimum number of arguments is one character argument, and the returned value is character.

```
int rtn4(m1,c1,p1,m2,c2,p2)
short *m1,*c1;
ptr *p1;
short *m2,*c2;
ptr *p2;
{
```

To find out the number of arguments actually specified by the user calling the function, use the FNCN routine as follows:

```
FNCN(narg)
```

The supervisor stores the number of arguments in `narg`. Remember to use `&narg` in C.

After you know the number of arguments, you can access the values of the arguments by calling FNCARG.

```
FNCARG(arg-number, &type-arg, &data-value-ptr, &current-len-ptr, &max-len);
```

When you call this routine, specify in *arg-number* the number of the argument whose value you are retrieving (this number should always be less than or equal to `narg`, returned by FNCN). The call to FNCARG returns these values:

- The *type-arg* contains the type of the argument: 1 if numeric, 2 if character.
- If *type-arg* is 1, the *data-value-ptr* contains a pointer to the double-precision floating-point value of the numeric argument.
- If *type-arg* is 2, the *data-value-ptr* contains a pointer to the character string and *current-len-ptr* contains a pointer to the current length of the character string. The *max-len* contains the maximum length of the character string if you change the value pointed to by *data-value-ptr* and need to reset the value pointed to by *current-len-ptr*.

Note: The *data-value-ptr* and *current-len-ptr* arguments to FNCARG must be addresses of pointers so that you can update values of arguments for CALL routines. You specify the address of the pointer, and the pointer value is changed to point to the value. You can then dereference the pointer to obtain or change the value pointed to.

The following example illustrates how to access the values of the arguments passed to the routine. The call to FNCN returns the number of arguments. The `for` loop obtains the values and uses the `memcpy` function to copy them to a character array. The length of the return string is set to the accumulated length of the character strings (or the maximum length if the accumulated length is too long), and the concatenated text strings are moved to the return value.

```
int rtn4(m1,c1,p1,m2,c2,p2)
short *m1,*c1;
ptr *p1;
short *m2,*c2;
ptr *p2;
{
short *cptr,maxlen;
ptr p,q;
double value,*valuep;
int i,l,n,type;
char retval[200];

   /*-----get argument count----------------------------------------*/
FNCN(&n);

   /*-----loop through, obtaining values and concatenating----------*/
q = *p2;
for (i=1,l=0;i<=n;i++) {
    FNCARG(i,&type,&valuep,&cptr,&maxlen);
    memcpy(retval+l,valuep,*cptr);
    l += *cptr;
    }

   /*-----set return argument length--------------------------------*/
*c2 = MIN(l,*m2);

   /*-----copy value into argument----------------------------------*/
memcpy(*p2,retval,*c2);
return(F_OK);
}
```

Differences for CALL Routines

A CALL routine differs from a function in two ways: CALL routines have no return value, and any argument updated by the processing routine is passed back to the calling routine.

The following example shows a CALL routine that updates both values passed to it. The incoming numeric argument, *value*, specifies which letter to use for the value of the character argument; *value* is reset to 99. The character argument is set to one of the five letters A through E, depending on the incoming value of the numeric argument.

```
int rtn5(value,m2,c2,p2)
double *value;
short *m2;
short *c2;
ptr *p2;
```

```
{
int temp;
static char alpha[] = {'A','B','C','D','E'};
temp = (!(1 <= *value && *value <= 5)) ? 1 : *value;
*c2 = 1;
*value = 99;
**p2 = alpha[temp-1];
return(F_OK);
}
```

Required Termination Routine

After you have completed all of the routines to process functions or CALL routines in a package, you must include the IFFEXT routine for termination processing. In many cases, the IFFEXT routine you code in your program is as simple as follows:

```
void iffext() {}
```

If you have processing that you want to perform before your function or CALL routine package is terminated, you can include that processing in this routine. For example, if you have allocated memory using a native language statement instead of one of the SAS_X memory allocation functions, be sure to free the allocated memory in the IFFEXT routine.

Note: If you plan to write many IFFC packages, and you do not intend to perform any termination processing, you may prefer to define the IFFEXT routine in a separate source module and simply link this module to every IFFC package you write.

Complete Example

The following program illustrates the complete example explained in parts earlier in this chapter. Refer to Chapter 16, "Sample IFFCs in the FORTRAN, PL/I, and IBM 370 Assembler Languages," for illustrations of the same functions or CALL routines in other supported languages.

Listing 15.1
C Functions and CALL Routines Example

```
/*-----------------------------------------------------------------*/
/* NAME:         FUNCC                                             */
/* TYPE:         FORMAT                                            */
/* LANGUAGE:     C                                                 */
/* PURPOSE:      Example function package to demonstrate how to    */
/*               write SAS/Toolkit functions                       */
/* NOTES:        There is also a PL/I, FORTRAN, and IBM ASM/370    */
/*               version of this routine.                          */
/*-----------------------------------------------------------------*/

/*-----the one necessary #include file-----------------------------*/

#include <uwproc>

/*-----------------------------------------------------------------*/
/* The iffmai routine is required. It is called by the SAS supervisor*/
/* at least once for request=1 and exactly once for request 2.     */
/* Request 1 is to obtain the names and attributes of the functions. */
/* Request 2 is to obtain the addresses of the routines to be called.*/
/*-----------------------------------------------------------------*/
```

(continued on next page)

Listing 15.1
(continued)

```
ptr iffmai(request)
int *request;
{

/*-------------------------------------------------------------------*/
/* With request 1, we call certain routines. The first routine we    */
/* call is UWPINI. This routine is only necessary if you are going   */
/* to use SAS_ interface routines in your code. We do indeed in this */
/* example, so the UWPINI call is necessary. Ensure a 0 is passed to */
/* it. The next call made is to FNCDFS. We pass this routine the     */
/* number of functions being defined. For each function, we provide  */
/* a FNCDFN call. This supplies the function number, function name,  */
/* the minimum and maximum number of arguments, and the return type  */
/* (1=numeric 2=character 0='call' routine). Following the FNCDFN    */
/* call, a call is made to FNCDFA as many times as there are minimum */
/* arguments to the function we're defining. FNCDFA specifies the    */
/* function number, the argument number, and the argument type (1    */
/* or 2 like above). The last call made is FNCDFE whose return code  */
/* is returned to the supervisor. For request 2, only the call to    */
/* FNCFNE needs to be called, and its return code is returned to the */
/* supervisor. Any additional initialization code that your appli-   */
/* cation may need should go in the request=2 code.                  */
/* The function rtn1 corresponds to the definitions for function 1   */
/* rtn2 for function 2, etc.                                         */
/*-------------------------------------------------------------------*/

if (*request == 1) {
    UWPRCC(0);
    FNCDFS(5);

    /*-----1. TESTFADD: exactly 2 args, returning numeric------------*/
    FNCDFN(1,"TESTFADD",2,2,1);

    /*-----both arguments are numeric---------------------------------*/
    FNCDFA(1,1,1);
    FNCDFA(1,2,1);

    /*-----2. TESTFSUB: exactly 2 args, returning numeric------------*/
    FNCDFN(2,"TESTFSUB",2,2,1);

    /*-----both arguments are numeric---------------------------------*/
    FNCDFA(2,1,1);
    FNCDFA(2,2,1);

    /*-----3. TESTFSUB: exactly 2 args, returning character----------*/
    FNCDFN(3,"TESTFCHR",2,2,2);

    /*-----arg1 numeric, arg2 character-------------------------------*/
    FNCDFA(3,1,2);
    FNCDFA(3,2,1);

    /*-----4. TESTFVC: 1-10 args, returning character----------------*/
    FNCDFN(4,"TESTFVC ",1,10,2);

    /*-----all arguments are character--------------------------------*/
    FNCDFA(4,1,2);
    FNCDFA(4,2,2);
    FNCDFA(4,3,2);
    FNCDFA(4,4,2);
    FNCDFA(4,5,2);
    FNCDFA(4,6,2);
    FNCDFA(4,7,2);
    FNCDFA(4,8,2);
    FNCDFA(4,9,2);
    FNCDFA(4,10,2);

    /*-----5. TESTFCAL: 2 arguments, CALL routine--------------------*/
    FNCDFN(5,"TESTFCAL",2,2,0);

    /*-----arg1 numeric, arg2 character-------------------------------*/
    FNCDFA(5,1,1);
    FNCDFA(5,2,2);

    return(FNCDFE());
    }
 else if (*request == 2) {
    return(FNCFNE());
    }
}
```

```
/*----------------------------------------------------------------*/
/* Each function routine rtnx has a calling sequence dependent on the*/
/* way the function was defined with request 1. rtn1 corresponds to  */
/* the TESTFADD function. This function requires exactly two numeric */
/* arguments and returns a numeric value. The function simply adds   */
/* two numbers together. If either number is missing, we return a    */
/* missing value. Note also that the return argument is the third    */
/* one in the calling sequence; the return value for rtn1 is a status*/
/* code to indicate F_OK, F_EMISS (missing value), or F_IAF+n (where */
/* n is the argument number that's illegal).                          */
/*----------------------------------------------------------------*/

int rtn1(arg1,arg2,arg3)
double *arg1,*arg2,*arg3;
{
if (SAS_ZMISS(*arg1) || SAS_ZMISS(*arg2))
   return(F_EMISS);
*arg3 = *arg1 + *arg2;
return(F_OK);
}

/*----------------------------------------------------------------*/
/* rtn2 is just like rtn1, except that we subtract the second arg    */
/* from the first. Once again we check for missing.                   */
/*----------------------------------------------------------------*/

int rtn2(arg1,arg2,arg3)
double *arg1,*arg2,*arg3;
{
if (SAS_ZMISS(*arg1) || SAS_ZMISS(*arg2))
   return(F_EMISS);
*arg3 = *arg1 - *arg2;
return(F_OK);
}

/*----------------------------------------------------------------*/
/* rtn3 has first a character argument, then a numeric argument,     */
/* and returns a character value. For character arguments, we use    */
/* a triple of arguments, that point to the maximum length, the      */
/* current length, and the pointer to the actual data. rtn3 corres-  */
/* ponds to TESTFCHR(a,b), where a is a character string, and b is   */
/* a number. All occurrences of a dash (-) in 'a' will be translated */
/* to the character in the delimiter list that corresponds to 'b'.   */
/* The delimiter list is )|@#$%¬&*(, and 'b' is the one-based index  */
/* into that list. For example, | would correspond to b=2.           */
/*----------------------------------------------------------------*/

int rtn3(m1,c1,p1,value,mr,cr,pr)
short *m1,*c1;
ptr *p1;
double *value;
short *mr,*cr;
ptr *pr;
{
int j;
ptr p,q;
char i;

/*-----our delimiters-----------------------------------------------*/
static char delims[10] = {')','|','@','#','$','%','¬','&','*','('};

/*-----get the index into the delimiter string------------------------*/
i = delims[((int)*value)-1];

/*-----set the current length of the returning character string------*/
*cr = *c1;

/*-----roving pointer variables----------------------------------------*/
p = *p1;
q = *pr;

/*-----loop through and change the dash to the other character-------*/
for (j = *c1;j>0;j--,p++,q++) {
    if ((*q = *p) == '-')
       *q = i;
    }

return(F_OK);
}
```

(continued on next page)

Listing 15.1 (continued)

```
/*-----------------------------------------------------------------*/
/* rtn4 demonstrates how to handle a varying number of arguments.   */
/* This routine corresponds to the TESTFVC(x1,x2,....x10) function.  */
/* This function simply concatenates all arguments together into a   */
/* resultant string. Since we defined TESTFVC to have a minimum of 1 */
/* argument, the triple of parameters for that argument must appear  */
/* in the rtn4 definition. And since it does return a character value*/
/* the triple for that value must also be present. All other args    */
/* (and even the first) can be accessed via the FNCARG routine. We   */
/* first call the FNCN routine to determine the actual number of      */
/* args passed by the function caller. We then obtain each one via   */
/* FNCARG. Once all concatenation is done, we can return.             */
/*-----------------------------------------------------------------*/

int rtn4(m1,c1,p1,m2,c2,p2)
short *m1,*c1;
ptr *p1;
short *m2,*c2;
ptr *p2;
{
short *cptr,maxlen;
ptr p,q;
double value,*valuep;
int i,l,n,type;
char retval[200];

/*-----get argument count------------------------------------------*/
FNCN(&n);

/*-----loop through, obtaining values and concatenating--------------*/
q = *p2;
for (i=1,l=0;i<=n;i++) {
    FNCARG(i,&type,&valuep,&cptr,&maxlen);
    memcpy(retval+l,valuep,*cptr);
    l += *cptr;
    }

/*-----set return argument length------------------------------------*/
*c2 = MIN(l,*m2);

/*-----copy value into argument--------------------------------------*/
memcpy(*p2,retval,*c2);
return(F_OK);
}

/*-----------------------------------------------------------------*/
/* rtn5 demonstrates how a call routine differs. rtn5 corresponds    */
/* to TESTFCAL(a,b), where a is a numeric variable and b is a        */
/* character variable. In our example, We set b to one of the        */
/* characters A,B,C,D,E based on the numeric value. If it's 1-5,     */
/* we set b to the matching character. Otherwise, we set it to 'A'.  */
/* We also reset the numeric variable a to 99.                       */
/* Note that callers of TESTFCAL should ONLY pass variables. The     */
/* routine cannot distinguish between variables and constants. If    */
/* a constant is passed, it will get changed, and can alter the      */
/* outcome of the DATA step in very confusing ways. If an value      */
/* resulting from an expression is passed, TESTFCAL will appear to   */
/* have no effect, since the values passed are stored in temporary   */
/* locations.                                                        */
/*-----------------------------------------------------------------*/

int rtn5(value,m2,c2,p2)
double *value;
short *m2;
short *c2;
ptr *p2;
{
int temp;
static char alpha[] = {'A','B','C','D','E'};

/*-----determine base 1 offset to use--------------------------------*/
temp = (!(1 <= *value && *value <= 5)) ? 1 : *value;

/*-----character argument now has a length of 1----------------------*/
*c2 = 1;

/*-----numeric argument now has a value of 99------------------------*/
*value = 99;

/*-----set single byte of character argument to proper character-----*/
**p2 = alpha[temp-1];
return(F_OK);
}
```

Chapter 16 Sample IFFCs in the FORTRAN, PL/I, and IBM® 370 Assembler Languages

Introduction

This chapter illustrates the PL/I, FORTRAN, and IBM 370 assembler versions of the informats discussed in Chapter 13, "Writing a SAS Informat," the formats discussed in Chapter 14, "Writing a SAS Format," and the functions and CALL routines discussed in Chapter 15, "Writing a SAS Function or CALL Routine."

Sample Informats

The following sections illustrate sample informats written in the PL/I, FORTRAN, and IBM 370 assembler languages.

PL/I Informats

The following program is the PL/I version of the informat illustrated in Chapter 13. Note that the PL/I IFFCs are in separate source code modules so that they can be accessed as separate external routines. Other techniques for identifying separate external routines in a single source code module are not portable between different operating systems.

Listing 16.1
PL/I Informat Example, Part 1

```
/*-----------------------------------------------------------------*/
/* NAME:          INFMTP1 (part 1 of 3)                            */
/* TYPE:          INFORMAT                                         */
/* LANGUAGE:      PL/I                                             */
/* PURPOSE:       Example informat package to demonstrating writing */
/*                SAS/Toolkit informats                            */
/* NOTES:         There is also a C,    FORTRAN, and IBM ASM/370   */
/*                version of this routine.                         */
/*                This source with be linked into the UWUTESTI     */
/*                executable. This is necessary since all entities */
/*                defined herein start with 'TESTI' (excluding @ and */
/*                '$').                                            */
/*-----------------------------------------------------------------*/

%INCLUDE MACPORT6;

 /*-----------------------------------------------------------------*/
 /* The iffmai routine is required. It is called by the SAS supervisor*/
 /* at least once for request=1 and exactly once for request 2.      */
 /* Request 1 is to obtain the names and attributes of the informats. */
 /* Request 2 is to obtain the addresses of the routines to be called.*/
 /*-----------------------------------------------------------------*/

 IFFMAI: PROC(REQUEST) PORTPROC RETURNS(PTR);
 %INCLUDE UWPROC;
 %INCLUDE UWIFF;
 DCL  REQUEST FXL;
 DCL  BINARY BUILTIN;

 /*-----------------------------------------------------------------*/
 /* With request 1, we call certain routines. The first routine we   */
 /* call is UWPINI. This routine is only necessary if you are going  */
 /* to use SAS_ interface routines in your code. We do indeed in this */
 /* example, so the UWPINI call is necessary. Ensure a 0 is passed to */
 /* it. The next call made is to FMTDFS. We pass this routine the    */
 /* number of entities (IFFs) being defined. We then follow with as  */
 /* many FMTDFN calls as there are entities. With FMTDFN, we give the */
 /* entity number which will match with the corresponding rtn        */
 /* routine (e.g., entity 1 corresponds to rtn1). We also give the   */
 /* entity name (8-character padded string), the minimum/maximum/    */
 /* default width, and the mininum/maximum/default number of decimals.*/
 /* The last argument, the justification code, is not applicable to  */
 /* informats and will ignored. The last call is to FMTDFE, which    */
 /* signals the end of informat/format definition. Its return code is */
 /* returned to the supervisor.                                      */
 /* Note that FMTDFN is used both for formats and informats. Informats*/
 /* are specified with a leading '@'. Note also that informats and   */
 /* formats cannot be mixed in the same package.                     */
 /*-----------------------------------------------------------------*/

 IF (REQUEST = 1) THEN DO;
    CALL UWPRCP(BINARY(0));
    CALL FMTDFS(2);
    CALL FMTDFN(1,'@TESTIN ',1,16,8,0,0,10,0);
    CALL FMTDFN(2,'@$TESTIC',1,16,8,0,0, 0,0);
    RETURN(FMTDFE());
    END;

 /*-----------------------------------------------------------------*/
 /* With request 2, we will only need to call the FMTFNE routine,    */
 /* which will pass to the supervisor the necessary info concerning  */
 /* the addresses of the rtnx routines. We need to return the return */
 /* code of FMTFNE. Note that if you have any kind of initialization */
 /* code for some external process, this is the place to put it.     */
 /*-----------------------------------------------------------------*/

 ELSE IF (REQUEST = 2) THEN DO;
    RETURN(FMTFNE());
    END;
 END IFFMAI;
```

Listing 16.2
PL/I Informat Example, Part 2

```
/*---------------------------------------------------------------*/
/* NAME:          INFMTP2 (part 1 of 3)                          */
/* TYPE:          INFORMAT                                       */
/* LANGUAGE:      PL/I                                           */
/* PURPOSE:       Example informat package to demonstrating writing */
/*                SAS/Toolkit informats                          */
/* NOTES:         There is also a C,    FORTRAN, and IBM ASM/370 */
/*                version of this routine.                       */
/*                This source with be linked into the UWUTESTI   */
/*                executable. This is necessary since all entities */
/*                defined herein start with 'TESTI' (excluding @ and */
/*                '$').                                          */
/*---------------------------------------------------------------*/

/*---------------------------------------------------------------*/
/* rtn1 is the first informat  we are defining. Note that all infmts */
/* will use the same calling sequence: the first argument is the  */
/* pointer to the input string,the second is the input            */
/* width, the third is the width of the output data, fourth is the */
/* number of decimals (always ignored for character), and fifth is */
/* the pointer to the output field. This last pointer is a dblptr */
/* for numeric informats, and a straight ptr for character informats.*/
/* This first informat routine, rtn1, corresponds to TESTIN. This in-*/
/* format will shift down each numeric character in the input string */
/* and then read in the new number. For example, '234' will become */
/* '123' and be read in as 123. Note that the return code from the */
/* SAS_XFXIN is returned to the supervisor. In general, any nonzero */
/* return code means the informat was not successful.             */
/*---------------------------------------------------------------*/

%INCLUDE MACPORT6;
 RTN1: PROC(FROM,INW,W,D,TO) RETURNS(FXL) PORTPROC;
 %INCLUDE UWPROC;
 DCL  FROM PTR;                    /* I - PTR TO DATA TO FORMAT     */
 DCL  (INW,                        /* I - WIDTH OF INPUT DATA       */
       W,                          /* I - WIDTH OF OUTPUT DATA      */
       D) FXL;                     /* I - NO. OF DECIMALS (ALWAYS 0)*/
 DCL  TO FTL;                      /* O - RESULTING VALUE           */

 DCL  TEMP CHAR(200);
 DCL  TEMPB CHAR(200) BASED;
 DCL  (TRANSLATE,SUBSTR) BUILTIN;

 /*-----copy into work area------------------------------------------*/
 TEMP = SUBSTR(FROM->TEMPB,1,INW);

 /*-----translate each digit to next lower digit----------------------*/
 TEMP = TRANSLATE(TEMP,'1234567890','0123456789');

 /*-----input the new string into the returning number location-------*/
 RETURN(SAS_XFXIN(ADDR(TEMP),INW,D,0,ADDR(TO)));
 END RTN1;
```

Listing 16.3
PL/I Informat Example, Part 3

```
/*---------------------------------------------------------------*/
/* NAME:          INFMTP3 (part 3 of 3)                          */
/* TYPE:          INFORMAT                                       */
/* LANGUAGE:      PL/I                                           */
/* PURPOSE:       Example informat package to demonstrating writing */
/*                SAS/Toolkit informats                          */
/* NOTES:         There is also a C,    FORTRAN, and IBM ASM/370 */
/*                version of this routine.                       */
/*                This source with be linked into the UWUTESTI   */
/*                executable. This is necessary since all entities */
/*                defined herein start with 'TESTI' (excluding @ and */
/*                '$').                                          */
/*---------------------------------------------------------------*/

%INCLUDE MACPORT6;

/*---------------------------------------------------------------*/
/* rtn2 corresponds to the informat $TESTIC. This informat will   */
/* attempt a small table lookup. If ABCDEFGH is seen, it will return */
/* PART 1. If SECONDVL is seen, it will return PART 2. If THIRDVL  */
/* is seen, it will return PART 3. If none of these is seen, it will */
/* return NOT FOUND.                                              */
/*---------------------------------------------------------------*/


RTN2: PROC(FROM,INW,W,D,TO) RETURNS(FXL);
DCL  FROM PTR;                     /* I - PTR TO DATA TO FORMAT     */
```

(continued on next page)

Listing 16.3 (continued)

```
DCL  (INW,                                /* I - WIDTH OF INPUT DATA       */
      W,                                  /* I - WIDTH OF OUTPUT DATA      */
      D) FXL;                             /* I - NO. OF DECIMALS (ALWAYS 0)*/
DCL  TO PTR;                              /* O - RESULTING VALUE           */
DCL  I FXL;
DCL  SUBSTR BUILTIN;

/*-----input values----------------------------------------------------*/
DCL  TABLE(3) CHAR(8) STATIC INIT('ABCDEFGH','SECONDVL','THIRDVL ');

/*-----output values---------------------------------------------------*/
DCL  RESP(4) CHAR(8) STATIC INIT('PART 1','PART 2','PART 3',
                                 'NOTFOUND');
DCL  TEMP CHAR(200) BASED;

/*-----look for a match in the tables----------------------------------*/
DO I=1 TO 3 WHILE(SUBSTR(FROM->TEMP,1,INW)¬=TABLE(I));
   END;

/*-----copy the proper table entry into place--------------------------*/
SUBSTR(TO->TEMP,1,W) = RESP(I);

RETURN(0);
END RTN2;
```

FORTRAN Informats

The following program is the FORTRAN version of the informat illustrated in Chapter 13, “Writing a SAS Informat.”

Listing 16.4 FORTRAN Informat Example

```
C *-----------------------------------------------------------------*
C * NAME:          INFMTF                                            *
C * TYPE:          FORMAT                                            *
C * PURPOSE:       Example informat package to demonstrating writing *
C *                SAS/Toolkit informats                             *
C * NOTES:         There is also a PL/I, C, and IBM ASM/370          *
C *                version of this routine.                          *
C *                This source with be linked into the UWUTESTI      *
C *                executable. This is necessary since all entities  *
C *                defined herein start with 'TESTI' (excluding @ and *
C *                '$').                                             *
C *-----------------------------------------------------------------*
C
C *-----------------------------------------------------------------*
C * The iffmai routine is required. It is called by the SAS supervisor*
C * at least once for request=1 and exactly once for request 2.      *
C * Request 1 is to obtain the names and attributes of the informats. *
C * Request 2 is to obtain the addresses of the routines to be called.*
C *-----------------------------------------------------------------*
C
      INTEGER FUNCTION IFFMAI*4 (IREQ)
      INCLUDE (PROTOSF)
      INTEGER*4 IREQ
C
C *-----------------------------------------------------------------*
C * With request 1, we call certain routines. The first routine we   *
C * call is UWPINI. This routine is only necessary if you are going  *
C * to use SAS_ interface routines in your code. We do indeed in this *
C * example, so the UWPINI call is necessary. Ensure a 0 is passed to *
C * it. The next call made is to FMTDFS. We pass this routine the    *
C * number of entities (IFFs) being defined. We then follow with as  *
C * many FMTDFN calls as there are entities. With FMTDFN, we give the *
C * entity number which will match with the corresponding rtn        *
C * routine (e.g., entity 1 corresponds to rtn1). We also give the   *
C * entity name (8-character padded string), the minimum/maximum/    *
C * default width, and the mininum/maximum/default number of decimals.*
C * The last argument, the justification code, is not applicable to  *
C * informats and will ignored. The last call is to FMTDFE, which    *
C * signals the end of informat/format definition. Its return code is *
C * returned to the supervisor.                                      *
C * Note that FMTDFN is used both for formats and informats. Informats*
C * are specified with a leading '@'. Note also that informats and   *
C * formats cannot be mixed in the same package.                     *
C *-----------------------------------------------------------------*
C
      IF (IREQ .NE. 1) GOTO 100
         CALL UWPRCF(0)
         CALL FMTDFS(2)
         CALL FMTDFN(1,'@TESTIN ',1,16,8,0,10,0,0)
         CALL FMTDFN(2,'@$TESTIC',1,16,8,0,0, 0,0)
```

```
      IFFMAI = FMTDFE(0)
      RETURN
C
C *-----------------------------------------------------------------*
C * With request 2, we will only need to call the FMTFNE routine,   *
C * which will pass to the supervisor the necessary info concerning *
C * the addresses of the rtnx routines. We need to return the return *
C * code of FMTFNE. Note that if you have any kind of initialization *
C * code for some external process, this is the place to put it.   *
C *-----------------------------------------------------------------*
C
100  IF (IREQ .NE. 2) GOTO 200
      IFFMAI = FMTFNE(0)
      RETURN
200  RETURN
     END
C
C *-----------------------------------------------------------------*
C * rtn1 is the first informat  we are defining. Note that all infmts *
C * will use the same calling sequence: the first argument is the   *
C * pointer to the input string,the second is the input             *
C * width, the third is the width of the output data, fourth is the *
C * number of decimals (always ignored for character), and fifth is  *
C * the pointer to the output field. This last pointer is a dblptr   *
C * for numeric informats, and a straight ptr for character informats.*
C * This first informat routine, rtn1, corresponds to TESTIN. This in-*
C * format will shift down each numeric character in the input string *
C * and then read in the new number. For example, '234' will become  *
C * '123' and be read in as 123. Note that the return code from the  *
C * SAS_XFXIN is returned to the supervisor. In general, any nonzero *
C * return code means the informat was not successful.               *
C *-----------------------------------------------------------------*
C
     INTEGER FUNCTION RTN1*4 (FROM,INW,W,D,TO)
     INTEGER*4 FROM,INW,W,D
     INCLUDE (PROTOSF)
     CHARACTER*1 TEMP(33)
     REAL*8 TO
C
     CALL SAS_ZMOVEI(FROM,IADDR(TEMP),INW)
     CALL SAS_ZSTRANS(IADDR(TEMP),INW,CADDR('1234567890'),
    +                 CADDR('0123456789'),10)
     RTN1 = SAS_XFXIN(IADDR(TEMP),INW,D,0,IADDR(TO))
     RETURN
     END
C
C *-----------------------------------------------------------------*
C * rtn2 corresponds to the informat $TESTIC. This informat will     *
C * attempt a small table lookup. If ABCDEFGH is seen, it will return *
C * PART 1. If SECONDVL is seen, it will return PART 2. If THIRDVL    *
C * is seen, it will return PART 3. If none of these is seen, it will *
C * return NOT FOUND.                                                *
C *-----------------------------------------------------------------*
C
     INTEGER FUNCTION RTN2*4 (FROM,INW,W,D,TO)
     INTEGER*4 FROM,INW,W,D,TO
     INCLUDE (PROTOSF)
     CHARACTER*8 TABLE(3),RESP(4)
C
C *-----input values-------------------------------------------------*
     DATA TABLE(1)/'ABCDEFGH'/
     DATA TABLE(2)/'SECONDVL'/
     DATA TABLE(3)/'THIRDVL '/
C
C *-----output values------------------------------------------------*
     DATA RESP(1) /'PART 1  '/
     DATA RESP(2) /'PART 2  '/
     DATA RESP(3) /'PART 3  '/
     DATA RESP(4) /'NOTFOUND'/
     CHARACTER*1 BLANK
     DATA BLANK/' '/
C
C *-----------------------------------------------------------------*
C * Note here the use of SAS_ZSTRCOM and SAS_ZSTRMOV. The SAS_ZSTRCOM *
C * routine will compare two fields with differing lengths, by assum- *
C * ing blank padding for the shorter. It returns 0 if a match, and  *
C * nonzero otherwise. SAS_ZSTRMOV moves from one location to another *
C * using the specified widths. If the destination length is longer, *
C * it pads with blanks. If shorter, it truncates.                   *
C *-----------------------------------------------------------------*
C
C *-----look for a match in the tables--------------------------------*
     DO 301 I=1,3
        IF (SAS_ZSTRCOM(FROM,INW,IADDR(TABLE(I)),8) .EQ. 0) GOTO 400
301     CONTINUE
400  CONTINUE
C
C *-----copy the proper table entry into place------------------------*
     CALL SAS_ZSTRMOV(IADDR(RESP(I)),8,TO,W)
```

(continued on next page)

Listing 16.4 (continued)

```
        RTN2 = 0
        RETURN
        END
```

IBM 370 Assembler Informats

The following program is the IBM 370 assembler version of the informat illustrated in Chapter 13, "Writing a SAS Informat."

Listing 16.5 IBM 370 Assembler Informat Example

```
*---------------------------------------------------------------------*
* NAME:         INFMTA                                                *
* TYPE:         FORMAT                                                *
* LANGUAGE:     IBM ASM/370                                           *
* PURPOSE:      Example format package to demonstrate how to write    *
*               SAS/Toolkit formats                                   *
* NOTES:        There is also a C, PL/I, and FORTRAN                  *
*               version of this routine.                              *
*               This is not reentrant.                                *
*               All return values are placed into R15 upon return.    *
*---------------------------------------------------------------------*
*
         SASREGS
*
*-----the single include file needed---------------------------------*
         COPY XFMT
*
*-----this one included since we may SAS_ calls---------------------*
         COPY UWPROC
         USING IFFMAI,R11
*
*---------------------------------------------------------------------*
* The iffmai routine is required. It is called by the SAS supervisor*
* at least once for request=1 and exactly once for request 2.        *
* Request 1 is to obtain the names and attributes of the informats. *
* Request 2 is to obtain the addresses of the routines to be called.*
*---------------------------------------------------------------------*
*
IFFMAI   CSECT
         B     SAVE-*(R15)               branch to standard prolog
         L     R1,0(R1)                  R1=request ptr
         L     R1,0(R1)                  R1=request
*
*---------------------------------------------------------------------*
* With request 1, we use certain macros. The first macro used is     *
* XFMTDFS. We pass this macro the number of entities (IFFs) being    *
* defined. We then follow with as many XFMTDFN invocations as there *
* entities. With XFMTDFN, we give the                                *
* entity number which will match with the corresponding rtn          *
* routine (e.g., entity 1 corresponds to rtn1). We also give the     *
* entity name, the minimum/maximum/default                           *
* width, and the mininum/maximum/default number of decimals. The     *
* last argument, the justification code, is unused with informats.   *
* The last macro invocation is to XFMTDFE, which signals the         *
* end of informat/format definition.                                 *
* Its return code is returned to the supervisor.                     *
*---------------------------------------------------------------------*
*
         C     R1,=F'1'                  if (request == 1)
         BNE   TRY2
         XC    PARMLIST(4),PARMLIST
         LA    R1,PARMLIST
         L     R15,=V(UWPRCC)
         BALR  R14,R15
         XFMTDFS 2
         XFMTDFN 1,@TESTIN,1,16,8,0,10,0,0
         XFMTDFN 2,@$TESTIC,1,16,8,0,0,0,0
         XFMTDFE
         B     RETURN                       return via epilog
*
*---------------------------------------------------------------------*
* With request 2, we will only need to invoke the XFMTFNE macro,     *
* which will pass to the supervisor the necessary info concerning    *
* the addresses of the rtnx routines. The return code of XFMTFNE     *
* will be set in R15. Note that if there is any kind if initiali-    *
* zation code for some external process, put it here.                *
*---------------------------------------------------------------------*
*
TRY2     C     R1,=F'2'                  if (request == 2)
         BNE   UNDEF
         XFMTFNE
```

```
         B      RETURN                        return via epilog
UNDEF    SLR    R15,R15                   otherwise unknown
         B      RETURN
*
*----------------------------------------------------------------*
* rtn1 is the first informat  we are defining. Note that all infmts *
* will use the same calling sequence: the first argument is the    *
* pointer to the input string,the second is the input              *
* width, the third is the width of the output data, fourth is the  *
* number of decimals (always ignored for character), and fifth is  *
* the pointer to the output field. This last pointer is a dblptr   *
* for numeric informats, and a straight ptr for character informats.*
* This first informat routine, rtn1, corresponds to TESTIN. This in-*
* format will shift down each numeric character in the input string *
* and then read in the new number. For example, '234' will become  *
* '123' and be read in as 123. Note that the return code from the  *
* SAS_XFXIN is returned to the supervisor. In general, any nonzero *
* return code means the informat was not successful.               *
*----------------------------------------------------------------*
*
         ENTRY RTN1
RTN1     DS     0H
         B      SAVE-*(R15)               to standard prolog
         LM     R1,R5,0(R1)               get args
         ST     INW,INWX                  save inw (r2) value
*-----copy into work area-------------------------------------------*
* SAS_ZMOVEI(from,temp,inw);
*
         BCTR   INW,0                     -1 for EX
         EX     INW,INTOTEMP              move data into temp
*-----translate each digit to next lower digit----------------------*
* SAS_ZSTRANS(temp,inw,"1234567890","0123456789",10);
*
         LA     R14,TEMP
         L      R15,INWX
         LA     R0,DIGITS+1
         LA     R1,DIGITS
         LA     R2,10
         STM    R14,R2,PARMLIST
         LA     R1,PARMLIST
         L      R15,&SAS_ZSTRANS
         BALR   R14,R15
*-----input the new string into the returning number location-------*
* return(SAS_XFXIN(temp,inw,d,0,to));
         LA     R14,TEMP
         L      R15,INWX
         LR     R0,D
         SLR    R1,R1
         LR     R2,TO
         STM    R14,R2,PARMLIST
         LA     R1,PARMLIST
         L      R15,&SAS_XFXIN
         BALR   R14,R15
         B      RETURN                    (r15 is already what we want)
DIGITS   DC     C'01234567890'
*
*----------------------------------------------------------------*
* rtn2 corresponds to the informat $TESTIC. This informat will     *
* attempt a small table lookup. If ABCDEFGH is seen, it will return *
* PART 1. If SECONDVL is seen, it will return PART 2. If THIRDVL   *
* is seen, it will return PART 3. If none of these is seen, it will *
* return NOT FOUND.                                                *
*----------------------------------------------------------------*
*
         ENTRY RTN2
RTN2     DS     0H
         B      SAVE-*(R15)               branch to standard prolog
         LM     R1,R5,0(R1)               get args
         MVI    TEMP,C' '                 set TEMP to all blanks
         MVC    TEMP+1(199),TEMP
         BCTR   R2,0
         EX     R2,INTOTEMP               copy into temp
*
*-----loop through looking for a match------------------------------*
         SLR    R15,R15
         LA     R1,3
         SLR    R2,R2
NEXTITEM LA     R4,TABLE(R2)              r4->table entry
         CLC    TEMP(8),0(R4)             does it match?
         BE     *+4+4+4
         LA     R2,8(R2)                  no - try next one
         BCT    R1,NEXTITEM
*
*-----pad with blanks into target-----------------------------------*
         MVI    0(TO),C' '                first char blank
         LA     R1,W
         BCTR   R1,0                      -1 for ex
         BCTR   R1,0                      -1 for blank propagation
         LTR    R1,R1                     any extra blanks to pad?
```

(continued on next page)

Listing 16.5 (continued)

```
         BM    *+4+4
         EX    R1,BLANKTO              yes - go add them in
*
*-----copy proper table entry into place-----------------------------*
         BCTR  W,0                     -1 for ex
         CH    W,=H'7'                 w=min(w,8-1)
         BNH   *+4+4
         LA    W,7
         LA    R2,RESULT(R2)           r2->proper string
         EX    W,TOTARGET              copy into target
         SLR   R15,R15                 return code = 0
         B     RETURN                  to standard epilog
BLANKTO  MVC   1(0,TO),0(TO)
TOTARGET MVC   0(0,TO),0(R2)
*
*-----input values---------------------------------------------------*
TABLE    DC    CL8'ABCDEFGH'
         DC    CL8'SECONDVL'
         DC    CL8'THIRDVL'
*-----output values--------------------------------------------------*
RESULT   DC    CL8'PART 1'
         DC    CL8'PART 2'
         DC    CL8'PART 3'
         DC    CL8'NOTFOUND'
*-----standard prolog/epilog-----------------------------------------*
SAVE     DS    0H                       standard prolog
         STM   R14,R12,12(R13)          save regs
         BALR  R11,0
         LA    R0,*-IFFMAI
         SLR   R11,R0                   set base reg to CSECT
         LA    R2,SAVEAREA
         ST    R2,8(R13)                chain in
         ST    R13,4(R2)                        saveareas
         LR    R13,R2
         B     4(R15)                   back to our code
*
RETURN   DS    0H                       standard prolog
         L     R13,SAVEAREA+4           get caller's savearea
         L     R14,12(R13)              restore all but R15
         LM    R0,R12,20(R13)
         BR    R14                      go back
         LTORG
*
INTOTEMP MVC   TEMP(0),0(FROM)
TEMP     DS    CL200                    workarea
PARMLIST DS    5F                       parmlist for SAS_ calls
INWX     DS    F                        input width
SAVEAREA DS    18F
         DS    F                        extra fullword needed by SAS/C
*
IFFEXT   CSECT                          dummy csect needed
         BR    R14
         END
```

Sample Formats

The following sections illustrate samples of formats written in the PL/I, FORTRAN, and IBM 370 assembler languages.

PL/I Formats

The following program is the PL/I version of the format illustrated in Chapter 14. Note that the PL/I IFFCs are in separate source code modules so that they can be accessed as separate external routines. Other techniques for identifying separate external routines in a single source code module are not portable between different operating systems.

Listing 16.6
PL/I Format Example, Part 1

```
/*---------------------------------------------------------------*/
/* NAME:          FMTP1 (part 1 of 3)                            */
/* TYPE:          FORMAT                                         */
/* LANGUAGE:      PL/I                                           */
/* PURPOSE:       Example format package to demonstrate how to write */
/*                SAS/Toolkit formats                            */
/* NOTES:         There is also a FORTRAN, C, and  IBM ASM/370   */
/*                version of this routine.                       */
/*---------------------------------------------------------------*/

/*-----member to be included before the PROC statement---------------*/

%INCLUDE MACPORT6;
IFFMAI: PROC(REQUEST) PORTPROC RETURNS(PTR);

/*-----members to be included after the PROC statement---------------*/

 %INCLUDE UWPROC;
 %INCLUDE UWIFF;

/*---------------------------------------------------------------*/
/* The iffmai routine is required. It is called by the SAS supervisor*/
/* at least once for request=1 and exactly once for request 2.    */
/* Request 1 is to obtain the names and attributes of the formats. */
/* Request 2 is to obtain the addresses of the routines to be called.*/
/*---------------------------------------------------------------*/


DCL  REQUEST FIXED BIN(31);
DCL  BINARY BUILTIN;

/*---------------------------------------------------------------*/
/* With request 1, we call certain routines. The first routine we   */
/* call is UWPINI. This routine is only necessary if you are going  */
/* to use SAS_ interface routines in your code. We do indeed in this */
/* example, so the UWPINI call is necessary. Ensure a 0 is passed to */
/* it. The next call made is to FMTDFS. We pass this routine the    */
/* number of entities (IFFs) being defined. We then follow with as  */
/* many FMTDFN calls as there are entities. With FMTDFN, we give the */
/* entity number which will match with the corresponding rtn        */
/* routine (e.g., entity 1 corresponds to rtn1). We also give the   */
/* entity name (8-character padded string), the minimum/maximum/    */
/* default width, and the mininum/maximum/default number of decimals,*/
/* and the justification code (0=left 1=right). The last call made  */
/* is to FMTDFE, which signals the end of informat/format definition.*/
/* Its return code is returned to the supervisor.                   */
/*---------------------------------------------------------------*/


IF (REQUEST = 1) THEN DO;
   CALL UWPRCP(BINARY(0));
   CALL FMTDFS(2);
   CALL FMTDFN(1,'$ONEOFF ',1,200,1,0,0,0,0);
   CALL FMTDFN(2,'ONEOFX  ',1,16,8,0,10,0,0);
   RETURN(FMTDFE());
   END;

/*---------------------------------------------------------------*/
/* With request 2, we will only need to call the FMTFNE routine,    */
/* which will pass to the supervisor the necessary info concerning  */
/* the addresses of the rtnx routines. We need to return the return */
/* code of FMTFNE. Note that if you have any kind of initialization */
/* code for some external process, this is the place to put it.     */
/*---------------------------------------------------------------*/


ELSE IF (REQUEST = 2) THEN DO;
   RETURN(FMTFNE());
   END;
END IFFMAI;
```

Listing 16.7
PL/I Format Example, Part 2

```
/*-------------------------------------------------------------------*/
/* NAME:         FMTP2 (part 2 of 3)                                 */
/* TYPE:         FORMAT                                              */
/* LANGUAGE:     PL/I                                                */
/* PURPOSE:      Example format package to demonstrate how to write  */
/*               SAS/Toolkit formats                                 */
/* NOTES:        There is also a FORTRAN, C, and  IBM ASM/370        */
/*               version of this routine.                            */
/*-------------------------------------------------------------------*/

/*-----member to be included before the PROC statement---------------*/

%INCLUDE MACPORT6;

/*-------------------------------------------------------------------*/
/* rtn1 is the first format    we are defining. Note that all formats*/
/* will use the same calling sequence: the first argument is the     */
/* pointer to the input data (note that this is a char * for char    */
/* formats, and a dblptr for numerics), the second is the input      */
/* width, the third is the width of the output data, fourth is the   */
/* number of decimals (always ignored for character), and fifth is  */
/* the pointer to the output field.                                  */
/* This first format   routine, rtn1, corresponds to $ONEOFF. This
/* format will shift each character in the input string to the next */
/* largest character. For example, 'ABC' would be formatted as       */
/* 'BCD'. It pads with blanks if necessary.                          */
/*-------------------------------------------------------------------*/


RTN1: PROC(FROM,INW,W,D,TO) RETURNS(FXL) PORTPROC;
%INCLUDE UWIFF;
DCL  FROM PTR;                    /* I - PTR TO DATA TO FORMAT     */
DCL  (INW,                        /* I - WIDTH OF INPUT DATA       */
      W,                          /* I - WIDTH OF OUTPUT DATA      */
      D) FXL;                     /* I - NO. OF DECIMALS (ALWAYS 0)*/
DCL  TO PTR;                      /* I - PTR TO OUTPUT DATA        */

DCL (I,J,K) FXL;
DCL  ARRAY(0:1) CHAR(1) BASED;
DCL  TEMPF FXS;
DCL  TEMPC(0:1) CHAR(1) BASED(ADDR(TEMPF));
DCL  (MIN,ADDR) BUILTIN;

/*-----shift each input byte, up to w bytes--------------------------*/

J = MIN(INW,W);
TEMPF = 0;
DO I=0 TO J-1 BY 1;
   TEMPC(LOW_ORDER_FXS) = FROM->ARRAY(I);
   TEMPF = TEMPF +1;
   TO->ARRAY(I) = TEMPC(LOW_ORDER_FXS);
   END;

/*-----pad the remainder with blanks---------------------------------*/

DO I=I TO W-1;
   TO->ARRAY(I) = ' ';
   END;

/*-----normal return code--------------------------------------------*/

RETURN(F_OK);

END RTN1;
```

Listing 16.8
PL/I Format Example, Part 3

```
/*-------------------------------------------------------------------*/
/* NAME:         FMTP3 (part 3 of 3)                                 */
/* TYPE:         FORMAT                                              */
/* LANGUAGE:     PL/I                                                */
/* PURPOSE:      Example format package to demonstrate how to write  */
/*               SAS/Toolkit formats                                 */
/* NOTES:        There is also a FORTRAN, C, and  IBM ASM/370        */
/*               version of this routine.                            */
/*-------------------------------------------------------------------*/

/*-----member to be included before the PROC statement---------------*/
%INCLUDE MACPORT6;

/*-------------------------------------------------------------------*/
/* rtn2 is the next  format we are defining. This format routine     */
/* corresponds to ONEOFX. This format will first format out the      */
/* input number normally (using the SAS_XFXPN standard formatting    */
```

```
/* routine). Then, it converts all leading blanks to #.             */
/*------------------------------------------------------------------*/

RTN2: PROC(FROM,INW,W,D,TO) PORTPROC RETURNS(FXL);
%INCLUDE UWPROC;
%INCLUDE UWIFF;
DCL  FROM FTL;                     /* I - VALUE TO FORMAT            */
DCL  (INW,                         /* I - WIDTH OF INPUT DATA        */
      W,                           /* I - WIDTH OF OUTPUT DATA       */
      D) FXL;                      /* I - NO. OF DECIMALS            */
DCL  TO PTR;                       /* I - PTR TO OUTPUT DATA         */

DCL  TEMP CHAR(32);
DCL  TARGET CHAR(200) BASED;
DCL  (SUBSTR,TRANSLATE) BUILTIN;
DCL  I FXL;

/*-----format normally----------------------------------------------*/
I = SAS_XFXPN(FROM,W,D,0,ADDR(TEMP));

/*-----translate blanks to #----------------------------------------*/
SUBSTR(TO->TARGET,1,W) = SUBSTR(TRANSLATE(TEMP,'#',' '),1,W);

RETURN(0);
END RTN2;
```

FORTRAN Formats

The following program is the FORTRAN version of the format illustrated in Chapter 14.

Listing 16.9
FORTRAN Format Example

```
C *-----------------------------------------------------------------*
C * NAME:          FMTF                                             *
C * TYPE:          FORMAT                                           *
C * LANGUAGE:      FORTRAN                                          *
C * PURPOSE:       Example format package to demonstrate how to write *
C *                SAS/Toolkit formats                              *
C * NOTES:         There is also a C, PL/I, and IBM ASM/370         *
C *                version of this routine.                         *
C *                This code must be preprocessed by the SAS/Toolkit *
C *                FTNPREP before being compiled.                   *
C *-----------------------------------------------------------------*
C
      INTEGER FUNCTION IFFMAI*4 (IREQ)
C
C *-----------------------------------------------------------------*
C * On IBM mainframes, we can use the INCLUDE feature. On other     *
C * systems, FTNPREP will expand this.                              *
C *-----------------------------------------------------------------*
C
      INCLUDE (PROTOSF)
      INTEGER*4 IREQ
C *-----------------------------------------------------------------*
C * The iffmai routine is required. It is called by the SAS supervisor*
C * at least once for request=1 and exactly once for request 2.     *
C * Request 1 is to obtain the names and attributes of the formats.  *
C * Request 2 is to obtain the addresses of the routines to be called.*
C *-----------------------------------------------------------------*
C
C *-----------------------------------------------------------------*
C * With request 1, we call certain routines. The first routine we  *
C * call is UWPINI. This routine is only necessary if you are going *
C * to use SAS_ interface routines in your code. We do indeed in this *
C * example, so the UWPINI call is necessary. Ensure a 0 is passed to *
C * it. The next call made is to FMTDFS. We pass this routine the   *
C * number of entities (IFFs) being defined. We then follow with as *
C * many FMTDFN calls as there are entities. With FMTDFN, we give the *
C * entity number which will match with the corresponding rtn       *
C * routine (e.g., entity 1 corresponds to rtn1). We also give the  *
C * entity name (8-character padded string), the minimum/maximum/   *
C * default width, and the mininum/maximum/default number of decimals,*
C * and the justification code (0=left 1=right). The last call made *
C * is to FMTDFE, which signals the end of informat/format definition.*
C * Its return code is returned to the supervisor.                  *
C *-----------------------------------------------------------------*
C
       IF (IREQ .NE. 1) GOTO 100
```

(continued on next page)

Listing 16.9 (continued)

```
        CALL UWPRCF(0)
        CALL FMTDFS(2)
        CALL FMTDFN(1,'$ONEOFF ',1,200,1,0,0,0,0)
        CALL FMTDFN(2,'ONEOFX  ',1,16,8,0,10,0,0)
        IFFMAI = FMTDFE(0)
        RETURN
C
C *-----------------------------------------------------------------*
C * With request 2, we will only need to call the FMTFNE routine,   *
C * which will pass to the supervisor the necessary info concerning *
C * the addresses of the rtnx routines. We need to return the return *
C * code of FMTFNE. Note that if you have any kind of initialization *
C * code for some external process, this is the place to put it.    *
C *-----------------------------------------------------------------*
C
 100  IF (IREQ .NE. 2) GOTO 200
        IFFMAI = FMTFNE(0)
        RETURN
 200  RETURN
      END
C
C *-----------------------------------------------------------------*
C * rtn1 is the first format    we are defining. Note that all formats*
C * will use the same calling sequence: the first argument is the   *
C * pointer to the input data (note that this is a char * for char  *
C * formats, and a dblptr for numerics), the second is the input    *
C * width, the third is the width of the output data, fourth is the *
C * number of decimals (always ignored for character), and fifth is *
C * the pointer to the output field.                                 *
C * This first format   routine, rtn1, corresponds to $ONEOFF. This
C * format will shift each character in the input string to the next *
C * largest character. For example, 'ABC' would be formatted as     *
C * 'BCD'. It pads with blanks if necessary.                         *
C *-----------------------------------------------------------------*
C
      INTEGER FUNCTION RTN1*4 (FROM,INW,W,D,TO)
      INTEGER*4 FROM,INW,W,D,TO
      INCLUDE (PROTOSF)
      LOGICAL*1 TEMP(200)
C
C *-----shift each input byte, up to w bytes--------------------------*
      CALL SAS_ZMOVEI(FROM,IADDR(TEMP),INW)
      DO 250 I=1,W
         TEMP(I) = LOG1I(ILOG1(TEMP(I))+1)
 250     CONTINUE
C
C *-----pad the remainder with blanks---------------------------------*
      CALL SAS_ZSTRMOV(IADDR(TEMP),INW,TO,W)
      RTN1 = 0
      RETURN
      END
C
C *-----------------------------------------------------------------*
C * rtn2 is the next  format we are definint. This format routine    *
C * corresponds to ONEOFX. This format will first format out the     *
C * input number normally (using the SAS_XFXPN standard formatting   *
C * routine). Then, it converts all leading blanks to #.             *
C *-----------------------------------------------------------------*
C
      INTEGER FUNCTION RTN2*4 (FROM,INW,W,D,TO)
      REAL*8 FROM
      INTEGER*4 INW,W,D,TO
      CHARACTER*1 TEMP(200)
      INCLUDE (PROTOSF)
C
C *-----format normally------------------------------------------------*
      I = SAS_XFXPN(FROM,W,D,0,IADDR(TEMP))
C
C *-----translate blanks to #------------------------------------------*
      DO 300 I=1,W
         IF (TEMP(I) .EQ. ' ')
     +      TEMP(I) = '#'
 300     CONTINUE
C
C *-----copy into target-----------------------------------------------*
      CALL SAS_ZMOVEI(IADDR(TEMP),TO,W)
      RTN2 = 0
      RETURN
      END
```

IBM 370 Assembler Formats

The following program is the IBM 370 assembler version of the format illustrated in Chapter 14.

Listing 16.10
IBM 370 Assembler Format Example

```
*---------------------------------------------------------------*
* NAME:        FMTA                                             *
* TYPE:        FORMAT                                           *
* LANGUAGE     IBM ASM/370                                      *
* PURPOSE:     Example format package to demonstrate how to write *
*              SAS/Toolkit formats                              *
* NOTES:       There is also a C, PL/I, and FORTRAN             *
*              version of this routine.                         *
*              This is not reentrant.                           *
*              All return values are placed into R15 upon return. *
*---------------------------------------------------------------*
*
         SASREGS
*
*-----the main  include file needed-----------------------------*
         COPY XFMT
*-----include file needed since SAS_ routines being called------*
         COPY UWPROC
         USING IFFMAI,R11
*
*---------------------------------------------------------------*
* The iffmai routine is required. It is called by the SAS supervisor*
* at least once for request=1 and exactly once for request 2.    *
* Request 1 is to obtain the names and attributes of the formats. *
* Request 2 is to obtain the addresses of the routines to be called.*
*---------------------------------------------------------------*
*
IFFMAI   CSECT
         B     SAVE-*(R15)            branch to standard prolog
         L     R1,0(R1)               R1=request ptr
         L     R1,0(R1)               R1=request
*
*---------------------------------------------------------------*
* With request 1, we use certain macros. The first macro used is *
* XFMTDFS. We pass this macro the number of entities (IFFs) being *
* defined. We then follow with as many XFMTDFN invocations as there *
* entities. With XFMTDFN, we give the                           *
* entity number which will match with the corresponding rtn     *
* routine (e.g., entity 1 corresponds to rtn1). We also give the *
* entity name, the minimum/maximum/default                      *
* width, and the mininum/maximum/default number of decimals,    *
* and the justification code (0=left 1=right). The last call made *
* is to FMTDFE, which signals the end of informat/format definition.*
* Its return code is returned to the supervisor.                *
*---------------------------------------------------------------*
*
         C     R1,=F'1'               if (request == 1)
         BNE   TRY2
         XC    PARMLIST(4),PARMLIST
         LA    R1,PARMLIST
         L     R15,=V(UWPRCC)
         BALR  R14,R15
         XFMTDFS 2
         XFMTDFN 1,$ONEOFF,1,200,1,0,0,0,0
         XFMTDFN 2,ONEOFX,1,32,1,0,0,0,0
         XFMTDFE
         B     RETURN                    return via epilog
*
*---------------------------------------------------------------*
* With request 2, we will only need to call the FMTFNE routine,  *
* which will pass to the supervisor the necessary info concerning *
* the addresses of the rtnx routines. We need to return the return *
* code of FMTFNE. Note that if you have any kind of initialization *
* code for some external process, this is the place to put it.  *
*---------------------------------------------------------------*
*
TRY2     C     R1,=F'2'               if (request == 2)
         BNE   UNDEF
         XFMTFNE
         B     RETURN                    return via epilog
UNDEF    SLR   R15,R15                otherwise unknown
         B     RETURN
         ENTRY RTN1
*
*---------------------------------------------------------------*
* rtn1 is the first format    we are defining. Note that all formats*
* will use the same calling sequence: the first argument is the  *
* pointer to the input data (note that this is a char * for char *
* formats, and a dblptr for numerics), the second is the input  *
* width, the third is the width of the output data, fourth is the *
* number of decimals (always ignored for character), and fifth is *
* the pointer to the output field.                              *
* This first format   routine, rtn1, corresponds to $ONEOFF. This *
* format will shift each character in the input string to the next *
* largest character. For example, 'ABC' would be formatted as   *
* 'BCD'. It pads with blanks if necessary.                      *
*---------------------------------------------------------------*
*
```

(continued on next page)

Listing 16.10
(continued)

```
RTN1     DS    0H
         B     SAVE-*(R15)           to standard prolog
         LM    R1,R5,0(R1)           gets args from parmlist
         LR    R0,INW                r0=inw
         CR    R0,W
         BNH   *+4+2
         LR    R0,W                  r0=min(inw,w)
*
*-----shift each input byte, up to w bytes--------------------------*
         SLR   R6,R6
         SLR   R7,R7
NXTCHAR  CR    R6,R0                 any more to shift?
         BNL   BLANKOUT
         IC    R7,0(R6,FROM)         r7=byte to drop down
         BCTR  R7,0                  drop down 1
         STC   R7,0(R6,TO)           save it back
         LA    R6,1(R6)              on to next byte
         B     NXTCHAR
*
*-----pad the remainder with blanks--------------------------------*
BLANKOUT DS    0H
         LA    R7,C' '               pad byte
RWEDONE  CR    R6,W                  any more to pad?
         BNL   DORET0
         STC   R7,0(R6,TO)           add in pad byte
         LA    R6,1(R6)              on to next byte
         B     RWEDONE
DORET0   SLR   R15,R15               return code of 0
         B     RETURN
*
*------------------------------------------------------------------*
* rtn2 is the next  format we are defining. This format routine    *
* corresponds to ONEOFX. This format will first format out the     *
* input number normally (using the SAS_XFXPN standard formatting   *
* routine). Then, it converts all leading blanks to #.             *
*------------------------------------------------------------------*
*
         ENTRY RTN2
RTN2     DS    0H
         B     SAVE-*(R15)           branch to standard prolog
         LM    R1,R5,0(R1)           get args
*
*-----format normally----------------------------------------------*
* SAS_XFXPN(*from,32,d,0,temp);
*
         MVC   PARMLIST(8),0(R1)
         MVC   PARMLIST+8,=F'32'
         ST    D,PARMLIST+12
         XC    PARMLIST+16(4),PARMLIST+16
         LA    R1,TEMP
         ST    R1,PARMLIST+20
         LA    R1,PARMLIST
         L     R15,&SAS_XFXPN
         BALR  R14,R15
*
*-----translate blanks to #----------------------------------------*
* SAS_ZSTRANC(temp,32,'#',' ');
*
         LA    R14,TEMP
         LA    R15,32
         LA    R0,C'#'
         LA    R1,C' '
         STM   R14,R1,PARMLIST
         LA    R1,PARMLIST
         L     R15,&SAS_ZSTRANC
         BALR  R14,R15
*
*-----copy into target---------------------------------------------*
* SAS_ZMOVEI(temp+32-w,to,w);
*
         LA    R1,TEMP+32            r1=temp+32-w
         SR    R1,W
         BCTR  W,0                   w-1 for EX
         EX    W,TOTARGET
         SLR   R15,R15               return code=0
         B     RETURN                to standard epilog
TOTARGET MVC   0(0,TO),0(R1)
*
SAVE     DS    0H                    standard prolog
         STM   R14,R12,12(R13)       save regs
         BALR  R11,0
         LA    R0,*-IFFMAI
         SLR   R11,R0                set base reg to CSECT
         LA    R2,SAVEAREA
         ST    R2,8(R13)             chain in
         ST    R13,4(R2)                    saveareas
         LR    R13,R2
         B     4(R15)                back to our code
*
RETURN   DS    0H                    standard prolog
         L     R13,SAVEAREA+4        get caller's savearea
         L     R14,12(R13)           restore all but R15
```

```
         LM    R0,R12,20(R13)
         BR    R14                         go back
         LTORG
*
*-----data areas-----------------------------------------------------*
TEMP     DS    32C                         area for rtn2 conversion
PARMLIST DS    6F                          parmlist for SAS_ calls
SAVEAREA DS    18F
         DS    F                           extra fullword needed by SAS/C
*
IFFEXT   CSECT                             dummy csect needed
         BR    R14
         END
```

Sample Test Programs for Formats and Informats

Here is a sample program that tests each of the formats and informats that have been defined in the sources given previously in this chapter:

```
*-----character format-----*;
data _null_;
     x=put('BCD',$oneoff3.);
     put 'SHOULD BE CDE: ' x=;
     run;

*-----numeric format-----*;
data _null_;
     x=put(123.45,oneofx10.);
     put 'SHOULD BE #######123: ' x=;
     run;

*-----character informat-----*;
data _null_;
     x=input('ABCDEFGH',$textic8.);
     put 'SHOULD BE PART 1: ' x=;
     run;

*-----numeric informat-----*;
data _null_;
     x=input('123.45',testin6.);
     put 'SHOULD BE 234.56: ' x=;
     run;
```

Sample Functions and CALL Routines

The following sections illustrate sample functions and CALL routines written in the PL/I, FORTRAN, and IBM 370 assembler languages.

PL/I Functions and CALL Routines

Writing a function or CALL routine in the PL/I language is somewhat more complicated than writing it in C because you must use %INCLUDE files in certain circumstances. Always use %INCLUDE MACPORT6 before the first line of code to initialize the various preprocessor symbols you need to use. Then, to make your program portable, you can declare variables using the preprocessor symbols in Table 16.1.

Table 16.1 *Portable Data Types for PL/I*

SAS/TOOLKIT Symbol	Standard Declaration	Data Type
FXL	FIXED BIN(31)	long integers
FXS	FIXED BIN(15)	short integers
FTL	FLOAT BIN(53)	double precision
FTS	FLOAT BIN(21)	single precision

Declare the IFFMAI routine or RTNx routine, using PORTPROC as an attribute in the PROC statement. This is so the code expands to use IBM PL/I Optimizing Compiler features on IBM mainframes.

After the PROC statement, include the UWPROC file if your function calls SAS_ routines. This member includes the necessary declarations for the SAS_X and SAS_Z routines. Always include the UWIFF file to ensure that the FNC and FMT routines and the preprocessor variables indicating return codes are declared.

The following program is the PL/I version of the functions and CALL routine illustrated in Chapter 15. Note that the PL/I IFFCs are in separate source code modules so that they can be accessed as separate external routines. Other techniques for identifying separate external routines in a single source code module are not portable between different operating systems.

Listing 16.11 *PL/I Function and CALL Routine Example, Part 1*

```
/*-----------------------------------------------------------------*/
/* NAME:          FUNCP1 (part 1 of 6)                             */
/* TYPE:          FUNCTION                                         */
/* LANGUAGE:      PL/I                                             */
/* PURPOSE:       Example function package to demonstrate how to   */
/*                write SAS/Toolkit functions                      */
/* NOTES:         There is also a C,   FORTRAN, and IBM ASM/370    */
/*                version of this routine.                         */
/*-----------------------------------------------------------------*/

%INCLUDE MACPORT6;

/*-----------------------------------------------------------------*/
/* The iffmai routine is required. It is called by the SAS supervisor*/
/* at least once for request=1 and exactly once for request 2.     */
/* Request 1 is to obtain the names and attributes of the functions. */
/* Request 2 is to obtain the addresses of the routines to be called.*/
/*-----------------------------------------------------------------*/

IFFMAI: PROC(REQUEST) PORTPROC RETURNS(PTR);
%INCLUDE UWPROC;
%INCLUDE UWIFF;
DCL  REQUEST FXL;
DCL  BINARY BUILTIN;

/*-----------------------------------------------------------------*/
/* With request 1, we call certain routines. The first routine we   */
/* call is UWPINI. This routine is only necessary if you are going  */
/* to use SAS_ interface routines in your code. We do indeed in this */
/* example, so the UWPINI call is necessary. Ensure a 0 is passed to */
/* it. The next call made is to FNCDFS. We pass this routine the    */
```

```
/* number of functions being defined. For each function, we provide  */
/* a FNCDFN call. This supplies the function number, function name,  */
/* the minimum and maximum number of arguments, and the return type  */
/* (1=numeric 2=character 0='call' routine). Following the FNCDFN    */
/* call, a call is made to FNCDFA as many times as there are minimum */
/* arguments to the function we're defining. FNCDFA specifies the    */
/* function number, the argument number, and the argument type (1    */
/* or 2 like above). The last call made is FNCDFE whose return code  */
/* is returned to the supervisor. For request 2, only the call to    */
/* FNCFNE needs to be called, and its return code is returned to the */
/* supervisor. Any additional initialization code that your appli-   */
/* cation may need should go in the request=2 code.                  */
/* The function rtn1 corresponds to the definitions for function 1   */
/* rtn2 for function 2, etc.                                         */
/*-------------------------------------------------------------------*/

IF (REQUEST = 1) THEN DO;
   CALL UWPRCP(BINARY(0));
   CALL FNCDFS(5);

   /*-----1. TESTFADD: exactly 2 args, returning numeric------------*/
   CALL FNCDFN(1,'TESTFADD',2,2,1);

   /*-----both arguments are numeric--------------------------------*/
   CALL FNCDFA(1,1,1);
   CALL FNCDFA(1,2,1);

   /*-----2. TESTFSUB: exactly 2 args, returning numeric------------*/
   CALL FNCDFN(2,'TESTFSUB',2,2,1);

   /*-----both arguments are numeric--------------------------------*/
   CALL FNCDFA(2,1,1);
   CALL FNCDFA(2,2,1);

   /*-----3. TESTFSUB: exactly 2 args, returning character----------*/
   CALL FNCDFN(3,'TESTFCHR',2,2,2);

   /*-----arg1 numeric, arg2 character------------------------------*/
   CALL FNCDFA(3,1,2);
   CALL FNCDFA(3,2,1);

   /*-----4. TESTFVC: 1-20 args, call routine-----------------------*/
   CALL FNCDFN(4,'TESTFVC ',0,20,0);
   CALL FNCDFA(4,1,1);
   CALL FNCDFA(4,2,2);
   CALL FNCDFA(4,3,1);
   CALL FNCDFA(4,4,2);
   CALL FNCDFA(4,5,1);
   CALL FNCDFA(4,6,2);
   CALL FNCDFA(4,7,1);
   CALL FNCDFA(4,8,2);
   CALL FNCDFA(4,9,1);
   CALL FNCDFA(4,10,3);

   /*-----5. TESTFCAL: 1-10 arguments, all numeric, call routine----*/
   CALL FNCDFN(5,'TESTFCAL',1,500,0);
   CALL FNCDFA(5,1,1);
   CALL FNCDFA(5,2,1);
   CALL FNCDFA(5,3,1);
   CALL FNCDFA(5,4,1);
   CALL FNCDFA(5,5,1);
   CALL FNCDFA(5,6,1);
   CALL FNCDFA(5,7,1);
   CALL FNCDFA(5,8,1);
   CALL FNCDFA(5,9,1);
   CALL FNCDFA(5,10,1);
   RETURN(FNCDFE());
   END;
ELSE IF (REQUEST = 2) THEN DO;
   RETURN(FNCFNE());
   END;
END IFFMAI;
```

Listing 16.12
PL/I Function and CALL Routine Example, Part 2

```
/*-------------------------------------------------------------------*/
/* NAME:         FUNCP2 (part 2 of 6)                                */
/* TYPE:         FUNCTION                                            */
/* LANGUAGE:     PL/I                                                */
/* PURPOSE:      Example function package to demonstrate how to      */
/*               write SAS/Toolkit functions                         */
/* NOTES:        There is also a C,    FORTRAN, and IBM ASM/370      */
/*               version of this routine.                            */
```

(continued on next page)

***Listing 16.12* (continued)**

```
/*-----------------------------------------------------------------*/

/*-----------------------------------------------------------------*/
/* Each function routine rtnx has a calling sequence dependent on the*/
/* way the function was defined with request 1. rtn1 corresponds to */
/* the TESTFADD function. This function requires exactly two numeric */
/* arguments and returns a numeric value. The function simply adds   */
/* two numbers together. If either number is missing, we return a    */
/* missing value. Note also that the return argument is the third    */
/* one in the calling sequence; the return value for rtn1 is a status*/
/* code to indicate F_OK, F_EMISS (missing value), or F_IAF+n (where */
/* n is the argument number that's illegal).                         */
/*-----------------------------------------------------------------*/

%INCLUDE MACPORT6;
 RTN1: PROC(ARG1,ARG2,ARG3) PORTPROC RETURNS(FXL);
 %INCLUDE UWPROC;
 %INCLUDE UWIFF;
 DCL  (ARG1,ARG2,ARG3) FLOAT BIN(53);

 IF (SAS_ZMISS(ARG1) ¬= 0 | SAS_ZMISS(ARG2) ¬= 0)
    THEN RETURN(F_EMISS);
 ARG3 = ARG1+ARG2;
 RETURN(F_OK);
 END RTN1;
```

Listing 16.13* *PL/I Function and CALL Routine Example, Part 3

```
/*-----------------------------------------------------------------*/
/* NAME:         FUNCP3 (part 3 of 6)                               */
/* TYPE:         FUNCTION                                           */
/* LANGUAGE:     PL/I                                               */
/* PURPOSE:      Example function package to demonstrate how to     */
/*               write SAS/Toolkit functions                        */
/* NOTES:        There is also a C,    FORTRAN, and IBM ASM/370     */
/*               version of this routine.                           */
/*-----------------------------------------------------------------*/

/*-----------------------------------------------------------------*/
/* rtn2 is just like rtn1, except that we subtract the second arg    */
/* from the first. Once again we check for missing.                  */
/*-----------------------------------------------------------------*/

%INCLUDE MACPORT6;
RTN2: PROC(ARG1,ARG2,ARG3) RETURNS(FXL) PORTPROC;
%INCLUDE UWPROC;
%INCLUDE UWIFF;
DCL  (ARG1,ARG2,ARG3) FTL;

IF (SAS_ZMISS(ARG1) ¬= 0 | SAS_ZMISS(ARG2) ¬= 0)
   THEN RETURN(F_EMISS);
ARG3 = ARG1-ARG2;
RETURN(F_OK);
END RTN2;
```

Listing 16.14* *PL/I Function and CALL Routine Example, Part 4

```
/*-----------------------------------------------------------------*/
/* NAME:         FUNCP4 (part 4 of 6)                               */
/* TYPE:         FUNCTION                                           */
/* LANGUAGE:     PL/I                                               */
/* PURPOSE:      Example function package to demonstrate how to     */
/*               write SAS/Toolkit functions                        */
/* NOTES:        There is also a C,    FORTRAN, and IBM ASM/370     */
/*               version of this routine.                           */
/*-----------------------------------------------------------------*/

/*-----------------------------------------------------------------*/
/* rtn3 has first a character argument, then a numeric argument,     */
/* and returns a character value. For character arguments, we use    */
/* a triple of arguments, that point to the maximum length, the      */
/* current length, and the pointer to the actual data. rtn3 corres-  */
/* ponds to TESTFCHR(a,b), where a is a character string, and b is   */
/* a number. All occurrences of a dash (-) in 'a' will be translated */
/* to the character in the delimiter list that corresponds to 'b'.   */
/* The delimiter list is )|@#$%¬&*(, and 'b' is the one-based index */
/* into that list. For example, | would correspond to b=2.           */
/*-----------------------------------------------------------------*/

%INCLUDE MACPORT6;
 RTN3: PROC(M1,C1,P1,VALUE,MR,CR,PR) RETURNS(FXL) PORTPROC;
 %INCLUDE UWIFF;
 DCL  (M1,MR) FXS;             /* MAXIMUM LENGTHS */
```

```
DCL  (C1,CR) FXS;          /* CURRENT LENGTHS */
DCL  (P1,PR) PTR;
DCL  VALUE FTL;
DCL  I FXL;
DCL  TEMP CHAR(200) BASED;
DCL  (TRANSLATE,SUBSTR) BUILTIN;

/*-----our delimiters-----------------------------------------------*/
DCL  DELIMS CHAR(10) STATIC INIT(')|@#$%¬&*(');

/*-----get the index into the delimiter string-----------------------*/
I = VALUE;

/*-----set the current length of the returning character string------*/
CR = C1;

/*-----change the dash to the other character------------------------*/
SUBSTR(PR->TEMP,1,CR) = TRANSLATE(SUBSTR(P1->TEMP,1,C1),
                                  SUBSTR(DELIMS,I,1),'-');
RETURN(F_OK);
END RTN3;
```

Listing 16.15
PL/I Function and CALL Routine Example, Part 5

```
%INCLUDE MACPORT6;
 RTN4: PROC PORTPROC RETURNS(FXL);
 %INCLUDE UWIFF;
 DCL VALUE FTL;
 DCL MORE FTL BASED(MOREPTR);
 DCL (MOREPTR,DUMMY) PTR;
 DCL CURLEN FXS BASED(CURLENPTR);
 DCL MAXLEN FXS;
 DCL (P,CURLENPTR) PTR;
 DCL STG CHAR(200) BASED(P);
 DCL (I,N,L) FXL;
 DCL (TYPE1,TYPE2,RETVAL) FXL;
 DCL (MIN,SUBSTR) BUILTIN;

 /*-----get argument count-------------------------------------------*/
 CALL FNCN(N);

 /*-----loop through, obtaining values--------------------------------*/
 DO I=1 TO N BY 2;
    CALL FNCARG(I,TYPE1,MOREPTR,CURLENPTR,MAXLEN);
    CALL FNCARG(I+1,TYPE2,P,CURLENPTR,MAXLEN);
    IF (TYPE1 ¬= 1)
       THEN RETURN(F_EIAF+I);
    IF (TYPE2 ¬= 2)
       THEN RETURN(F_EIAF+I+1);
    L = MIN(MORE,CURLEN);
    SUBSTR(STG,1,L)=TRANSLATE(SUBSTR(STG,1,L),
            'BCDEFGHIJKLMNOPQRSTUVWXYZA',
            'ABCDEFGHIJKLMNOPQRSTUVWXYZ');
    MORE = MORE + 10;
    END;
 RETURN(F_OK);
 END RTN4;
```

Listing 16.16
PL/I Function and CALL Routine Example, Part 6

```
/*----------------------------------------------------------------*/
/* This test function allows from 1 to n  numeric arguments, and    */
/* will set each one to 10+ the value.                              */
/*----------------------------------------------------------------*/

%INCLUDE MACPORT6;
 RTN5: PROC(VALUE)          PORTPROC RETURNS(FXL);
  /* changed uwiff include file */
  DCL FMTDFS    ENTRY(FXL);
  DCL FMTDFN    ENTRY(FXL,CHAR(8),FXL,FXL,FXL,FXL,FXL,FXL,FXL);
  DCL FMTDFE    ENTRY RETURNS(PTR);
  DCL FMTFNE    ENTRY RETURNS(PTR);
  DCL FNCDFS    ENTRY(FXL);
  DCL FNCDFN    ENTRY(FXL,CHAR(8),FXL,FXL,FXL);
  DCL FNCDFA    ENTRY(FXL,FXL,FXL);
  DCL FNCDFE    ENTRY RETURNS(PTR);
  DCL FNCFNE    ENTRY RETURNS(PTR);
  DCL FNCN      ENTRY(FXL);
  DCL FNCARG    ENTRY(FXL,FXL,PTR,PTR,FXS); /* changed */
  %INCLUDE UWIFFH;
```

(continued on next page)

Listing 16.16
(continued)

```
DCL VALUE FTL;
DCL MORE FTL BASED(MOREPTR);
DCL MOREPTR PTR;
DCL C FXS;
DCL P PTR;
DCL (I,N) FXL;

/*-----get argument count-------------------------------------------*/
CALL FNCN(N);

VALUE = VALUE + 10;

/*-----loop through, obtaining values-------------------------------*/
DO I=2 TO N;
   CALL FNCARG(I,TYPE,MOREPTR,P,C);
   MORE = MORE + 10;
   END;
RETURN(F_OK);
END RTN5;
```

FORTRAN Functions

The following program is the FORTRAN version of the functions and CALL routine illustrated in Chapter 15.

Be sure to include the declarations for all FNC routines. On IBM systems, you can use INCLUDE (PROTOSF) to include all declarations. If you do not use a FORTRAN preprocessor, you must declare each of the FNC routines separately.

Listing 16.17 *FORTRAN Function and CALL Routine Example*

```
C *-----------------------------------------------------------------*
C * NAME:           FUNCF                                          *
C * TYPE:           FORMAT                                         *
C * LANGUAGE:       FORTRAN                                        *
C * PURPOSE:        Example function package to demonstrate how to *
C *                 write SAS/Toolkit functions                    *
C * NOTES:          There is also a C, PL/I, and IBM ASM/370       *
C *                 version of this routine.                       *
C *                 This code must be preprocessed by the SAS/Toolkit *
C *                 FTNPREP before being compiled.                 *
C *-----------------------------------------------------------------*
C
C *-----------------------------------------------------------------*
C * The iffmai routine is required. It is called by the SAS supervisor*
C * at least once for request=1 and exactly once for request 2.     *
C * Request 1 is to obtain the names and attributes of the functions. *
C * Request 2 is to obtain the addresses of the routines to be called.*
C *-----------------------------------------------------------------*
C
      INTEGER FUNCTION IFFMAI*4(IREQ)
C
C *-----------------------------------------------------------------*
C * On IBM mainframes, we can use the INCLUDE feature. On other     *
C * systems, FTNPREP will expand this.                              *
C *-----------------------------------------------------------------*
C
      INCLUDE (PROTOSF)
      INTEGER*4 IREQ
C
C *-----------------------------------------------------------------*
C * With request 1, we call certain routines. The first routine we  *
C * call is UWPINI. This routine is only necessary if you are going *
C * to use SAS_ interface routines in your code. We do indeed in this *
C * example, so the UWPINI call is necessary. Ensure a 0 is passed to *
C * it. The next call made is to FNCDFS. We pass this routine the   *
C * number of functions being defined. For each function, we provide *
C * a FNCDFN call. This supplies the function number, function name, *
C * the minimum and maximum number of arguments, and the return type *
C * (1=numeric 2=character 0='call' routine). Following the FNCDFN  *
C * call, a call is made to FNCDFA as many times as there are minimum *
C * arguments to the function we're defining. FNCDFA specifies the  *
C * function number, the argument number, and the argument type (1  *
C * or 2 like above). The last call made is FNCDFE whose return code *
C * is returned to the supervisor. For request 2, only the call to  *
C * FNCFNE needs to be called, and its return code is returned to the *
C * supervisor. Any additional initialization code that your appli- *
C * cation may need should go in the request=2 code.                *
C * The function rtn1 corresponds to the definitions for function 1 *
C * rtn2 for function 2, etc.                                       *
C *-----------------------------------------------------------------*
```

```
C
      IF (IREQ .NE. 1) GOTO 100
         CALL UWPRCF(0)
         CALL FNCDFS(5)
C
C *------------------1. TESTFADD: exactly 2 args, returning numeric---*
         CALL FNCDFN(1,'TESTFADD',2,2,1)
C
C *---------both arguments are numeric--------------------------------*
         CALL FNCDFA(1,1,1)
         CALL FNCDFA(1,2,1)
C
C *------------------2. TESTFSUB: exactly 2 args, returning numeric---*
         CALL FNCDFN(2,'TESTFSUB',2,2,1)
C
C *------------------both arguments are numeric-----------------------*
         CALL FNCDFA(2,1,1)
         CALL FNCDFA(2,2,1)
C
C *------------------3. TESTFCHR: exactly 2 args, returning character-*
         CALL FNCDFN(3,'TESTFCHR',2,2,2)
C
C *------------------arg1 numeric, arg2 character---------------------*
         CALL FNCDFA(3,1,2)
         CALL FNCDFA(3,2,1)
C
C *------------------4. TESTFVC: 1-10 args, returning character-------*
         CALL FNCDFN(4,'TESTFVC ',1,10,2)
C
C *------------------all arguments are character----------------------*
         CALL FNCDFA(4,1,2)
         CALL FNCDFA(4,2,2)
         CALL FNCDFA(4,3,2)
         CALL FNCDFA(4,4,2)
         CALL FNCDFA(4,5,2)
         CALL FNCDFA(4,6,2)
         CALL FNCDFA(4,7,2)
         CALL FNCDFA(4,8,2)
         CALL FNCDFA(4,9,2)
         CALL FNCDFA(4,10,2)
C
C *------------------5. TESTFCAL: 2 arguments, CALL routine-----------*
         CALL FNCDFN(5,'TESTFCAL',2,2,0)
C
C *------------------arg1 numeric, arg2 character---------------------*
         CALL FNCDFA(5,1,1)
         CALL FNCDFA(5,2,2)
         IFFMAI = FNCDFE(0)
         RETURN
  100 IF (IREQ .NE. 2) GOTO 200
         IFFMAI = FNCFNE(0)
         RETURN
  200 RETURN
      END
C
C *-------------------------------------------------------------------*
C * Each function routine rtnx has a calling sequence dependent on the*
C * way the function was defined with request 1. rtn1 corresponds to  *
C * the TESTFADD function. This function requires exactly two numeric *
C * arguments and returns a numeric value. The function simply adds   *
C * two numbers together. If either number is missing, we return a   *
C * missing value. Note also that the return argument is the third    *
C * one in the calling sequence; the return value for rtn1 is a status*
C * code to indicate F_OK, F_EMISS (missing value), or F_IAF+n (where *
C * n is the argument number that's illegal).                         *
C *-------------------------------------------------------------------*
C
      INTEGER FUNCTION RTN1*4(ARG1,ARG2,RETARG)
      INCLUDE (PROTOSF)
      REAL*8 ARG1,ARG2,RETARG
C
      IF (SAS_ZMISS(ARG1) .EQ. 0 .AND. SAS_ZMISS(ARG2) .EQ. 0) GOTO 10
         RTN1 = F_EMISS
         RETURN
   10 CONTINUE
C
      RETARG = ARG1+ARG2
      RTN1 = F_OK
      END
C
C *-------------------------------------------------------------------*
C * rtn2 is just like rtn1, except that we subtract the second arg    *
C * from the first. Once again we check for missing.                  *
C *-------------------------------------------------------------------*
C
      INTEGER FUNCTION RTN2*4(ARG1,ARG2,RETARG)
      INCLUDE (PROTOSF)
      REAL*8 ARG1,ARG2,RETARG
C
```

(continued on next page)

Listing 16.17
(continued)

```
      RTN2 = F_OK
      IF (SAS_ZMISS(ARG1) .EQ. 0 .AND. SAS_ZMISS(ARG2) .EQ. 0) GOTO 20
         RTN2 = F_EMISS
         RETURN
 20   CONTINUE
C
      RETARG = ARG1-ARG2
      RTN1 = F_OK
      END
C
C *-------------------------------------------------------------------*
C * rtn3 has first a character argument, then a numeric argument,     *
C * and returns a character value. For character arguments, we use    *
C * a triple of arguments, that point to the maximum length, the      *
C * current length, and the pointer to the actual data. rtn3 corres-  *
C * ponds to TESTFCHR(a,b), where a is a character string, and b is   *
C * a number. All occurrences of a dash (-) in 'a' will be translated *
C * to the character in the delimiter list that corresponds to 'b'.   *
C * The delimiter list is )|@#$%¬&*(, and 'b' is the one-based index  *
C * into that list. For example, | would correspond to b=2.           *
C *-------------------------------------------------------------------*
C
      INTEGER FUNCTION RTN3*4(M1,C1,P1,VALUE,MR,CR,PR)
      INTEGER*2 M1,C1,MR,CR
      INTEGER*4 P1,PR
      REAL*8 VALUE
      INTEGER*4 I,J
      CHARACTER*1 TEMP(200)
      CHARACTER*1 DELIM(10)
      CHARACTER*10 DELIMX
      EQUIVALENCE (DELIM,DELIMX)
C
C *-----our delimiters------------------------------------------------*
      DATA DELIMX/')|@#$%¬&*('/
C
C *-----get the index into the delimiter string-----------------------*
      I = VALUE
      J = C1
C
C *-----must move to TEMP since we can't reference actual data--------*
      CALL SAS_ZMOVEI(P1,IADDR(TEMP),J)
C
C *-----loop through and change the dash to the other character-------*
      DO 30 J=1,C1
         IF (TEMP(J) .EQ. '-')
     +      TEMP(J) = DELIM(I)
 30      CONTINUE
C
C *-----set current length based on max length------------------------*
      J = C1
      IF (J .GT. MR)
     +   J = MR
      CR = J
C
C *-----move to permanent location------------------------------------*
      CALL SAS_ZMOVEI(IADDR(TEMP),PR,J)
      RTN3 = F_OK
      END
C
C *-------------------------------------------------------------------*
C * rtn4 demonstrates how to handle a varying number of arguments.    *
C * This routine corresponds to the TESTFVC(x1,x2,....x10) function.  *
C * This function simply concatenates all arguments together into a  *
C * resultant string. Since we defined TESTFVC to have a minimum of 1 *
C * argument, the triple of parameters for that argument must appear  *
C * in the rtn4 definition. And since it does return a character value*
C * the triple for that value must also be present. All other args    *
C * (and even the first) can be accessed via the FNCARG routine. We   *
C * first call the FNCN routine to determine the actual number of     *
C * args passed by the function caller. We then obtain each one via   *
C * FNCARG. Once all concatenation is done, we can return.            *
C * A FORTRAN node about INTEGER*2 and INTEGER*4: The length values   *
C * passed to us are INTEGER*2. However, when we call SAS_ZMOVEI, it  *
C * expects INTEGER*4. Therefore, we need to set an INTEGER*4         *
C * variable before calling SAS_ZMOVEI.                               *
C *-------------------------------------------------------------------*
C
      INTEGER FUNCTION RTN4*4(M1,C1,P1,M2,C2,P2)
      INTEGER*2 M1,C1,M2,C2,C
      INTEGER*4 P1,P2,P
      REAL*8 VALUE
      CHARACTER*1 RETVAL(0:199)
C
C *-----get argument count--------------------------------------------*
      CALL FNCN(N)
C
C *-----loop through, obtaining values and concatenating--------------*
      L = 0
      DO 500 I=1,N
         CALL FNCARG(I,ITYPE,VALUE,C,P)
C
C ENSURE WE USE INTEGER*4 IN SAS_ZMOVEI CALL
```

```
          J = C
          CALL SAS_ZMOVEI(P,IADDR(RETVAL(L)),J)
          L = L + J
 500      CONTINUE
C
C *-----set return argument length----------------------------------*
      J = L
      IF (J .GT. M2)
     +   J = M2
      C2 = J
C
C ENSURE WE USE INTEGER*4 IN SAS_ZMOVEI CALL
C *-----copy value into argument-------------------------------------*
      CALL SAS_ZMOVEI(IADDR(RETVAL),P2,J)
      RTN4 = F_OK
      END
C
C *-----------------------------------------------------------------*
C * rtn5 demonstrates how a call routine differs. rtn5 corresponds   *
C * to TESTFCAL(a,b), where a is a numeric variable and b is a       *
C * character variable. In our example, We set b to one of the       *
C * characters A,B,C,D,E based on the numeric value. If it's 1-5,    *
C * we set b to the matching character. Otherwise, we set it to 'A'. *
C * We also reset the numeric variable a to 99.                      *
C * Note that callers of TESTFCAL should ONLY pass variables. The    *
C * routine cannot distinguish between variables and constants. If   *
C * a constant is passed, it will get changed, and can alter the     *
C * outcome of the DATA step in very confusing ways. If an value     *
C * resulting from an expression is passed, TESTFCAL will appear to  *
C * have no effect, since the values passed are stored in temporary  *
C * locations.                                                       *
C *-----------------------------------------------------------------*
C
      INTEGER FUNCTION RTN5*4(ARG1,M2,C2,P2)
      REAL*8 ARG1
      INTEGER*2 M2,C2
      INTEGER*4 P2
      CHARACTER*1 ALPHAX(5)
      CHARACTER*5 ALPHA5
      EQUIVALENCE (ALPHA5,ALPHAX)
      DATA ALPHA5/'ABCDE'/
C
C *-----determine base 1 offset to use--------------------------------*
      IF (.NOT.(1.0 .LE. ARG1 .AND. ARG1 .LE. 5)) GOTO 600
         I = ARG1
         GOTO 601
 600  I = 1
 601  CONTINUE
C
C *-----set single byte of character argument to proper character-----*
      CALL SAS_ZMOVEI(IADDR(ALPHAX(I)),P2,1)
C
C *-----numeric argument now has a value of 99------------------------*
      ARG1 = 99.0
C
C *-----character argument now has a length of 1----------------------*
      C2 = 1
      IF (C2 .GT. M2)
     +   C2 = M2
      RTN5 = F_OK
      END
```

IBM 370 Assembler Functions

The following program is the IBM 370 assembler version of the functions and CALL routine illustrated in Chapter 15.

Listing 16.18 *IBM 370 Assembler Function and CALL Routine Example*

```
*-----------------------------------------------------------------*  FUN00010
* NAME:          FUNCA                                            *  FUN00020
* TYPE:          FORMAT                                           *  FUN00030
* LANGUAGE:      IBM ASM/370                                      *  FUN00040
* PURPOSE:       Example function package to demonstrate how to   *  FUN00050
*                write SAS/Toolkit functions                      *  FUN00060
* NOTES:         There is also a C, PL/I, and FORTRAN             *  FUN00070
*                version of this routine.                         *  FUN00080
*-----------------------------------------------------------------*  FUN00110
*                                                                    FUN00120
```

(continued on next page)

Listing 16.18 (continued)

```
*-----------------------------------------------------------------*
* The iffmai routine is required. It is called by the SAS supervisor*
* at least once for request=1 and exactly once for request 2.      *
* Request 1 is to obtain the names and attributes of the functions. *
* Request 2 is to obtain the addresses of the routines to be called.*
*-----------------------------------------------------------------*
*
*-----macro to convert double in FR0 to integer in R1---------------*
         MACRO
&TAG     FLTTOINT
&TAG     AW    FR0,=XL8'4E00000100000000' convert to
         STD   FR0,DOUBLE                    integer
         L     R1,DOUBLE+4             r1=integer equivalent
         MEND
         SASREGS
         COPY  XFNC
         USING IFFMAI,R11
IFFMAI   CSECT
         B     SAVE-*(R15)
         L     R1,0(R1)
         L     R1,0(R1)
*
*-----------------------------------------------------------------*
* With request 1, we call certain routines. The first routine we   *
* call is UWPINI. This routine is only necessary if you are going  *
* to use SAS_ interface routines in your code. (We don't do this   *
* here, so it is not called. The first macro call is to XFNCDFS.   *
*                                        We pass this macro   the  *
* number of functions being defined. For each function, we provide *
* an XFNCDFN macro call. This supplies the  number, function name, *
* the minimum and maximum number of arguments, and the return type *
* (1=numeric 2=character 0='call' routine). Following the XFNCDFN  *
* call, a call is made to XFNCDFA as many times as there are minimum*
* arguments to the function we're defining. XFNCDFA specifies the  *
* function number, the argument number, and the argument type (1   *
* or 2 like above). The last macro is FNCDFE whose return code     *
* is returned to the supervisor. For request 2, only the call to   *
* XFNCFNE needs to be called, and its return code is returned to the*
* supervisor. Any additional initialization code that your appli-  *
* cation may need should go in the request=2 code.                 *
* The function rtn1 corresponds to the definitions for function 1  *
* rtn2 for function 2, etc.                                        *
*-----------------------------------------------------------------*
*
         C     R1,=F'1'
         BNE   TRY2
         XFNCDFS 5
*
*-----------------1. TESTFADD: exactly 2 args, returning numeric---*
         XFNCDFN 1,TESTFADD,2,2,1
*
*---------both arguments are numeric--------------------------------*
         XFNCDFA 1,1,1
         XFNCDFA 1,2,1
*
*-----------------2. TESTFSUB: exactly 2 args, returning numeric---*
         XFNCDFN 2,TESTFSUB,2,2,1
*
*-----------------both arguments are numeric-----------------------*
         XFNCDFA 2,1,1
         XFNCDFA 2,2,1
*
*-----------------3. TESTFCHR: exactly 2 args, returning character-*
         XFNCDFN 3,TESTFCHR,2,2,2
*
*-----------------arg1 numeric, arg2 character---------------------*
         XFNCDFA 3,1,2
         XFNCDFA 3,2,1
*
*-----------------4. TESTFVC: 1-10 args, returning character-------*
         XFNCDFN 4,TESTFVC,1,10,2
*
*-----------------all arguments are character----------------------*
         XFNCDFA 4,1,2
         XFNCDFA 4,2,2
         XFNCDFA 4,3,2
         XFNCDFA 4,4,2
         XFNCDFA 4,5,2
         XFNCDFA 4,6,2
         XFNCDFA 4,7,2
         XFNCDFA 4,8,2
         XFNCDFA 4,9,2
         XFNCDFA 4,10,2
*
*-----------------5. TESTFCAL: 2 arguments, CALL routine-----------*
         XFNCDFN 5,TESTFCAL,2,2,0
*
*-----------------arg1 numeric, arg2 character---------------------*
         XFNCDFA 5,1,1
         XFNCDFA 5,2,2
         XFNCDFE
         B     RETURN
```

```
TRY2     C      R1,=F'2'                                                FUN00980
         BNE    UNDEF                                                   FUN00990
         XFNCFNE                                                        FUN01000
         B      RETURN                                                  FUN01010
UNDEF    SLR    R15,R15                                                 FUN01020
         B      RETURN                                                  FUN01030
*                                                                       FUN01040
*-----------------------------------------------------------------*     FUN01050
* Each function routine rtnx has a calling sequence dependent on the*   FUN01060
* way the function was defined with request 1. rtn1 corresponds to  *   FUN01070
* the TESTFADD function. This function requires exactly two numeric *   FUN01080
* arguments and returns a numeric value. The function simply adds   *   FUN01090
* two numbers together. If either number is missing, we return a    *   FUN01100
* missing value. Note also that the return argument is the third    *   FUN01110
* one in the calling sequence; the return value for rtn1 is a status*   FUN01120
* code to indicate F_OK, F_EMISS (missing value), or F_IAF+n (where *   FUN01130
* n is the argument number that's illegal).                         *   FUN01140
*-----------------------------------------------------------------*     FUN01150
*                                                                       FUN01160
         ENTRY RTN1                                                     FUN01170
RTN1     DS     0H                                                      FUN01180
         B      SAVE-*(R15)              branch to standard prolog      FUN01190
         LM     R1,R3,0(R1)              get args                       FUN01200
         ISMISS 0(R1)                    ensure arg1 is nonmissing      FUN01210
         ISMISS 0(R2)                    ensure arg2 is nonmissing      FUN01220
         LD     FR0,0(R1)                FR0 = arg1                     FUN01230
         AD     FR0,0(R2)                FR0 += arg2                    FUN01240
         STD    FR0,0(R3)                arg3 = FR0                     FUN01250
         SLR    R15,R15                  indicate success               FUN01260
         B      RETURN                   to standard epilog             FUN01270
*                                                                       FUN01280
*-----------------------------------------------------------------*     FUN01290
* rtn2 is just like rtn1, except that we subtract the second arg    *   FUN01300
* from the first. Once again we check for missing.                  *   FUN01310
*-----------------------------------------------------------------*     FUN01320
*                                                                       FUN01330
         ENTRY RTN2                                                     FUN01340
RTN2     DS     0H                                                      FUN01350
         B      SAVE-*(R15)              branch to standard prolog      FUN01360
         LM     R1,R3,0(R1)              get args                       FUN01370
         ISMISS 0(R1)                    ensure arg1 is nonmissing      FUN01380
         ISMISS 0(R2)                    ensure arg2 is nonmissing      FUN01390
         LD     FR0,0(R1)                FR0 = arg1                     FUN01400
         SD     FR0,0(R2)                FR0 -= arg2                    FUN01410
         STD    FR0,0(R3)                arg3 = FR0                     FUN01420
         SLR    R15,R15                  indicate success               FUN01430
         B      RETURN                   to standard epilog             FUN01440
         EJECT                                                          FUN01450
*-----------------------------------------------------------------*     FUN01460
* rtn3 has first a character argument, then a numeric argument,     *   FUN01470
* and returns a character value. For character arguments, we use    *   FUN01480
* a triple of arguments, that point to the maximum length, the      *   FUN01490
* current length, and the pointer to the actual data. rtn3 corres-  *   FUN01500
* ponds to TESTFCHR(a,b), where a is a character string, and b is   *   FUN01510
* a number. All occurrences of a dash (-) in 'a' will be translated *   FUN01520
* to the character in the delimiter list that corresponds to 'b'.   *   FUN01530
* The delimiter list is )¦@#$%¬&*(, and 'b' is the one-based index  *   FUN01540
* into that list. For example, ¦ would correspond to b=2.           *   FUN01550
*-----------------------------------------------------------------*     FUN01560
*                                                                       FUN01570
         ENTRY RTN3                                                     FUN01580
RTN3     DS     0H                                                      FUN01580
         B      SAVE-*(R15)              do prolog
         MVC    PARMLIST(7*4),0(R1)      save original parms
*
*-----get the delimiter string to replace the dash-------------------*  FUN01730
*
         L      R1,PARMLIST+(4-1)*4      r1->numeric argument
         LD     FR0,0(R1)                fr0=numeric argument
         FLTTOINT                        convert to int in r1
         BCTR   R1,0                     -1 for base 0 offset
         IC     R0,DELIMS(R1)            r0=character to replace dash   FUN01710
*
*-----set the current length of the returning character string-------*  FUN01760
*
         L      R1,PARMLIST+(6-1)*4      r1->curlen of return arg
         L      R2,PARMLIST+(2-1)*4      r2->curlen of char arg
         L      R3,PARMLIST+(5-1)*4      r3->maxlen of return arg
         LH     R3,0(R3)                 r3=maxlen of return arg
         LH     R2,0(R2)                 r2=curlen of char arg
         CR     R2,R3                    is curlen(arg1)>maxlen(return)
         BL     *+4+2
         LR     R2,R3                       if so, use maxlen(return)
         STH    R2,0(R1)                 save that as return arg's
         LTR    R2,R2                    curlen not positive?
         BNP    WEREDONE                   no - we're done
*
*-----roving pointer variables----------------------------------------*  FUN01790
*
```

(continued on next page)

Listing 16.18 (continued)

```
         L     R1,PARMLIST+(3-1)*4      r1->char arg's dataptr
         L     R1,0(R1)                 r1->char arg's data
         L     R3,PARMLIST+(7-1)*4      r3->return arg's dataptr
         L     R3,0(R3)                 r3->char arg's data
*
*-----loop through and change the dash to the other character--------*  FUN01830
*
DASHLOOP DS    0H
         MVC   0(1,R3),0(R1)            copy over a character
         CLI   0(R3),C'-'               is it a dash?
         BNE   *+4+4
         STC   R0,0(R3)                    yes: replace with new delim
         LA    R1,1(R1)                 to next input character
         LA    R3,1(R3)                 to next output character
         BCT   R2,DASHLOOP              on to next character
WEREDONE DS    0H
         SLR   R15,R15                  indicate success
         B     RETURN                      and return
         ENTRY RTN4                                                     FUN01950
*                                                                       FUN01960
*--------------------------------------------------------------------*  FUN01970
* rtn4 demonstrates how to handle a varying number of arguments.     *  FUN01980
* This routine corresponds to the TESTFVC(x1,x2,....x10) function.   *  FUN01990
* This function simply concatenates all arguments together into a    *  FUN02000
* resultant string. Since we defined TESTFVC to have a minimum of 1  *  FUN02010
* argument, the triple of parameters for that argument must appear   *  FUN02020
* in the rtn4 definition. And since it does return a character value*  FUN02030
* the triple for that value must also be present. All other args     *  FUN02040
* (and even the first) can be accessed via the FNCARG routine. We    *  FUN02050
* first call the FNCN routine to determine the actual number of      *  FUN02060
* args passed by the function caller. We then obtain each one via    *  FUN02070
* FNCARG. Once all concatenation is done, we can return.             *  FUN02080
* A FORTRAN node about INTEGER*2 and INTEGER*4: The length values    *  FUN02090
* passed to us are INTEGER*2. However, when we call SAS_ZMOVEI, it   *  FUN02100
* expects INTEGER*4. Therefore, we need to set an INTEGER*4          *  FUN02110
* variable before calling SAS_ZMOVEI.                                *  FUN02120
* Note that in this example, we must be aware of the fact that the   *
* the total length of the concatenated values may exceed the maximum*
* allowable length for the return argument. If that is the case, we *
* must not fill in more than maxlen bytes.                           *
*--------------------------------------------------------------------*  FUN02130
*                                                                       FUN02140
RTN4     DS    0H                                                       FUN02150
         B     SAVE-*(R15)              std prolog                      FUN02160
         MVC   PARMLIST(6*4),0(R1)      copy in the incoming parmlist
         L     R2,PARMLIST+(6-1)*4      r2->return char dataptr
         L     R2,0(R2)                 r2->return char data
         L     R5,PARMLIST+(4-1)*4      r5->maxlen
         LH    R5,0(R5)                 r5=maxlen
**-----get argument count---------------------------------------------*  FUN02310
         LA    R1,N
         ST    R1,FNCPARMS              &n into FNCN parmlist
         LA    R1,FNCPARMS
         L     R15,=V(FNCN)             FNCN(&n);
         BALR  R14,R15
**-----loop through, obtaining values and concatenating---------------*  FUN02340
         LA    R0,1
         ST    R0,I                     I=1
         SLR   R4,R4                    zero accumulated length
         LA    R14,TYPE
         LA    R15,VALUEP
         LA    R0,CURLEN
         LA    R1,P
         STM   R14,R1,FNCPARMS+4        &type,&valuep,&curlen in plist
         L     R1,I
TOPLOOP  DS    0H                       DO I=1 TO N
         C     R1,N
         BH    ENDLOOP
         CR    R4,R5                       any more room in retchar?
         BNL   ENDLOOP
         ST    R1,I                        update I for next iteration
         ST    R1,FNCPARMS+0               save in plist as argnum
         LA    R1,FNCPARMS
         L     R15,=V(FNCARG)
         BALR  R14,R15                     call FNCARG to get info
         LH    R1,CURLEN                   r1=length of char arg
         LA    R0,0(R1,R4)                 r0=tot length with this arg
         SR    R0,R5                       r0=totlen-allowed len
         BNP   *+4+2+4
         SR    R1,R0                       r1=allowable bytes
         BNP   ENDLOOP
         L     R3,P                        r3->char arg's data
         BCTR  R1,0                        -1 for EX
         EX    R1,DOCONCAT                 concatenate the data
         LA    R2,1(R1,R2)                 skip over newly added data
         LA    R4,1(R1,R4)                 accumulate length
         L     R1,I
         LA    R1,1(R1)
         B     TOPLOOP                     END;
ENDLOOP  DS    0H
         L     R1,PARMLIST+(5-1)*4      r1->retarg's length
         STH   R4,0(R1)                 save new length
```

```
         SLR   R15,R15                  indicate success              FUN02170
         B     RETURN                     and return                  FUN02180
DOCONCAT MVC   0(0,R2),0(R3)
         EJECT                                                        FUN02490
*---------------------------------------------------------------*     FUN02500
* rtn5 demonstrates how a call routine differs. rtn5 corresponds  *   FUN02510
* to TESTFCAL(a,b), where a is a numeric variable and b is a      *   FUN02520
* character variable. In our example, We set b to one of the      *   FUN02530
* characters A,B,C,D,E based on the numeric value. If it's 1-5,   *   FUN02540
* we set b to the matching character. Otherwise, we set it to 'A'. *  FUN02550
* We also reset the numeric variable a to 99.                     *   FUN02560
* Note that callers of TESTFCAL should ONLY pass variables. The   *   FUN02570
* routine cannot distinguish between variables and constants. If  *   FUN02580
* a constant is passed, it will get changed, and can alter the    *   FUN02590
* outcome of the DATA step in very confusing ways. If an value    *   FUN02600
* resulting from an expression is passed, TESTFCAL will appear to *   FUN02610
* have no effect, since the values passed are stored in temporary *   FUN02620
* locations.                                                      *   FUN02630
*---------------------------------------------------------------*     FUN02640
*                                                                     FUN02650
         ENTRY RTN5                                                   FUN02660
RTN5     DS    0H                                                     FUN02670
         B     SAVE-*(R15)              standard prolog               FUN02680
         LM    R2,R5,0(R1)              get parmlist ptrs into regs:
*                                         r2->arg1 (numeric)
*                                         r3->m1 (maxlen of char arg)
*                                         r4->c1 (curlen of char arg)
*                                         r5->ptr1 (ptr to char data)
*-----determine base 0 offset to use---------------------------------* FUN02800
* temp = (!(1 <= *value && *value <= 5)) ? 1 : *value;                FUN02810
         LD    FR0,0(R2)                fr0 = numeric argument
         CD    FR0,=D'1'                if (fr0 < 1 or
         BL    *+4+4+4
         CD    FR0,=D'5'                    fr0 > 5)
         BNH   *+4+4
         LD    FR0,=D'1'                   fr0 = 1
         FLTTOINT ,                     convert fr0 to int in R1
         BCTR  R1,0                      to get base 0 offset for alpha
*
*-----character argument now has a length of 1-----------------------* FUN02830
*
         MVC   0(2,R4),=H'1'            save halfword 1
*
*-----numeric argument now has a value of 99-------------------------* FUN02860
*
         MVC   0(8,R2),=D'99'           save new double of 99
*
*-----set single byte of character argument to proper character------* FUN02890
*
         L     R5,0(R5)                 r5->char data
         LA    R1,ALPHA(R1)             r1->alpha character to use
         MVC   0(1,R5),0(R1)            store it into char value
*
*-----set zero return code and return--------------------------------* FUN02890
*
         SLR   R15,R15                  indicate success              FUN02690
         B     RETURN                   say goodbye                   FUN02700
*
*-----standard prolog/epilog----------------------------------------*  FUN02930
*
SAVE     DS    0H                       standard prolog               FUN02940
         STM   R14,R12,12(R13)          save regs                     FUN02950
         BALR  R11,0                                                  FUN02960
         LA    R0,*-IFFMAI                                            FUN02970
         SLR   R11,R0                   set base reg to CSECT         FUN02980
         LA    R2,SAVEAREA                                            FUN02990
         ST    R2,8(R13)                chain in                      FUN03000
         ST    R13,4(R2)                         saveareas            FUN03010
         LR    R13,R2                                                 FUN03020
         B     4(R15)                   back to our code              FUN03030
*                                                                     FUN03040
RETURN   DS    0H                       standard prolog               FUN03050
         L     R13,SAVEAREA+4           get caller's savearea         FUN03060
         L     R14,12(R13)              restore all but R15           FUN03070
         LM    R0,R12,20(R13)                                         FUN03080
         BR    R14                      go back                       FUN03090
         LTORG                                                        FUN03100
* NON-REENTRANT PORTION                                               FUN03110
DOUBLE   DS    D
SAVEAREA DS    18F                      standard savearea             FUN03120
         DS    F                        extra fullword needed by SAS/C FUN03130
PARMLIST DS    7A                       copy of incoming parmlist
FNCPARMS DS    5A                       parmlist for FNCN and FNCARG
I        DS    F                        incrementer for TESTFVC
N        DS    F                        nargs to TESTFVC
TYPE     DS    F                        type returned by FNCARG
VALUEP   DS    A                        dblptr returned by FNCARG
P        DS    A                        charptr returned by FNCARG
CURLEN   DS    H                        curlen returned by FNCARG
```

(continued on next page)

Listing 16.18
(continued)

```
* CONSTANTS
DELIMS   DC    C')|@#$%¬&&*('         delimiters for TESTFCHR       FUN01710
ALPHA    DC    C'ABCDE'               char list for TESTFCAL
IFFEXT   CSECT                                                      FUN03140
         BR    R14                                                  FUN03150
         END                                                        FUN03160
```

Sample Test Programs for Functions

The following test program shows how the functions are invoked:

```
data _null_;
     x=testfadd(1,2);
     put 'SHOULD BE 3: ' x=;
     run;
data _null_;
     x=testfsub(1,2);
     put 'SHOULD BE -1: ' x=;
     run;
data _null_;
     x=testfchr('A-B-C',2);
     put 'SHOULD BE A|B|C: ' x=;
     run;
data _null_;
     length x $200;
     x=testfvc('A','B','C');
     put 'SHOULD BE ABC: ' x=;
     run;

   /* missing value tests */
data _null_;
     x=testfadd(.,2);
     put 'SHOULD BE . (AND A MISSING VALUE MESSAGE): ' x=;
     run;
data _null_;
     x=testfsub(.,2);
     put 'SHOULD BE . (AND A MISSING VALUE MESSAGE): ' x=;
     run;

   /* CALL routine */
data _null_;
     X=2;
     Y='A';
     call testfcal(x,y);
     put 'SHOULD BE X=99 Y=b: ' x= y=;
     run;
```

Part 5
Converting Old Programs

Chapter 17 Converting from Version 5 to Version 6

Introduction

This chapter describes the differences between a Version 5 user-written procedure and a Version 6 procedure written in PL/I. This chapter focuses on changes to

PL/I procedures because that was the only language supported in Version 5. Included in the discussion are changes to

- □ the grammar commands
- □ the structures used by the SAS_X routines
- □ the process of checking return codes
- □ calls to SAS_X and SAS_Z routines (formerly the X and sublib routines).

The process for writing formats, informats, functions, and CALL routines is entirely different in Version 6, so no comparison is made for these types of user-written modules.

The comparisons in this chapter provide general information on the differences between Version 6 and Version 5. For detailed information on how to use the Version 6 routines, refer to the "Reference" division of this book.

Grammar Processing

Grammar processing in Version 6 is very similar to grammar processing in Version 5. Table 17.1 lists all the grammar codes available in Version 5 and shows which of them are available in Version 6.

Note: Table 17.1 does not show all the grammar codes that are available in Version 6. In particular, there are a number of new lexicals for date and time values. Refer to Chapter 21, "Grammar Language," for detailed information on the grammar codes that are now available.

Table 17.1 *Comparison of Version 5 and Version 6 Grammars*

Version 5 Grammar Element	Type of Element	Comparison with Version 6	Notes
ENDJB	lexical	same	
FMT	lexical	no equivalent	
FUNNAME	lexical	no equivalent	
INT	lexical	same	
NAME	lexical	same	
NBR	lexical	same	
QS	lexical	same	
BYVARLIST	production	some differences	Use BYSTMT to completely define the BY statement.
CHAR	production	no equivalent	
CHARLIST	production	no equivalent	Use NAMELIST or QS.
DOLIST	production	no equivalent	Use NUMLIST instead.
DSFIELD	production	same	

(continued)

Table 17.1 *(continued)*

Version 5 Grammar Element	Type of Element	Comparison with Version 6	Notes
NAMELIST	production	same	
NUMBER	production	same	
NUMLIST	production	same	
ONENAME	production	same	
ONEVAR	production	same	
VARLIST	production	same	
@DS	semantic action	some differences	You now specify eight arguments instead of three.
@DSDFLT	semantic action	some differences	You now specify eight arguments instead of three.
@FMT	semantic action	no equivalent	
@LABEL	semantic action	same	
@LIST	semantic action	same	
@LISTCT	semantic action	same	
@LISTVAL	semantic action	same	
@MARK	semantic action	same	
@OPT	semantic action	same	
@PARM	semantic action	same	
@PARMVAL	semantic action	same	
@PROCINIT	semantic action	same	
@RANGE	semantic action	same	
@RESETN	semantic action	no equivalent	
@SETDS	semantic action	no equivalent	Use @DSLKUP.
@STMTEND	semantic action	same	
@STMTINIT	semantic action	same	
@STMTLIST	semantic action	same	
@STMTPROC	semantic action	same	
@SPECIAL(1)	semantic action	same	
@SPECIAL(2)	semantic action	same	
@SPECIAL(4)	semantic action	no equivalent	
@SPECIAL(5)	semantic action	no equivalent	
@SPECIAL(6)	semantic action	no equivalent	
@SPECIAL(12)	semantic action	no equivalent	
@SPECIAL(1*nn*)	semantic action	same	

Structures

Many of the structures used in Version 5 are different or nonexistent in Version 6. Table 17.2 summarizes the differences in Version 6.

Note: Table 17.2 does not show all the structures available in Version 6. For a description of the new structures and detailed descriptions of the new versions of previously existing structures, refer to Chapter 20, "Data Structures."

Table 17.2 *Comparison of Version 5 and Version 6 Structures*

Version 5 Structure	Differences for Version 6
FATTRSTR	provides more data. The values are now FIXED BIN(15), not FIXED BIN(31).
GRAFSTR	no equivalent.
LABELSTR	no equivalent. Use the SAS_XINFO routine with the XO_LIB, XO_MEM, XO_TYPE, and XO_LABEL codes.
LINESTR	no equivalent structure, but the information provided by this structure is now available through calls to SAS_XPOPTGT and SAS_XPOPTST.
MACCONS	no equivalent. Refer to "Macro Variables" in Chapter 24, "Miscellaneous Routines and Values," for comparable symbols.
NAMESTR	fields have changed. The use of the NAMESTR is essentially the same as in Version 5. The NAMESTR can now store the informat length and number of decimal places, which was not possible in Version 5.
PROCPRS	now the PROC structure. The fields have changed. The primary use of the procedure is to obtain the HEAD pointer to the first statement structure, which is equivalent to the PROCPTR in Version 5.
STMTSTR	now the STMT structure. The fields have changed.
TOKEN	no equivalent.
XCOM	no equivalent.
XLINK	no equivalent.
XOPNSTR	fields have changed. In Version 6, you use the XCOPNSTR with utility files and the XOPNSTR only with SAS data sets.

General Comparison of Procedure Writing

Converting a Version 5 procedure to a Version 6 procedure involves a few changes to calling sequences, but there are relatively few new or different concepts. The Version 5 and Version 6 procedure environments are very similar. Both use grammars to parse procedure statements and store the results in the statement structures; both use similar file concepts to access SAS data sets and utility files; and both use the same scatter-read and gather-write processes for SAS data set I/O. Although Version 6 provides new techniques for printing output, most of the printing routines available in Version 5 are still available to PL/I programmers in Version 6. The following three sections describe some general differences in procedure writing.

Using Character Variables

In Version 6, PL/I character variables cannot be used directly in most calling sequences. Instead of passing a character variable, you must pass the address of the character variable. Version 6 routines expect pointer arguments. Use the ADDR function whenever you need to pass a character string. (Remember that the ADDR function cannot be given a constant argument.)

Note: The exceptions to this rule are the old-style printing routines (the ones that correspond to the Version 5 printing routines) and the SAS_XOPT routines. These routines can directly accept a character string.

Functions versus CALL Routines

Many of the X routines that were previously CALL routines are now functions. In most cases, the value returned is a return code that provides information about the success of the call to the function. In particular, the SAS_XC, SAS_XO, and SAS_XV routines return a special type of return code, RCTYPE. When a routine returns one of these codes, you can use the SAS_XPRLOG routine to print descriptive error messages that explain why a call to the routine failed. When you call SAS_XPRLOG routine with the RCTYPE value returned by a SAS_XC, SAS_XO, or SAS_XV routine, the SAS_XPRLOG routine evaluates the return code and prints messages only when the return code indicates the call was unsuccessful.

Summary of Changes in Routines

Table 17.3 lists all the X routines in Version 5 and indicates the corresponding SAS_X routine in Version 6. The last column of the table briefly notes the differences, if any, in using the Version 6 routine.

Note: This table does not show all the Version 6 routines that are available. To make the best use of Version 6, you should be familiar with all of the available SAS_X routines, which are described in detail in Chapter 22, "SAS_X Routines."

Table 17.3 *Comparison of Version 5 and Version 6 Routines*

Version 5 X Routine	Version 6 SAS_X Routine	Comparison	Notes
XBYLINE		no equivalent	
XBYLIST	SAS_XBYLIST	same	This is now a function, not a CALL routine.
XBYNEXT	SAS_XBYNEXT	same	
XBYSYNC	SAS_XBYSYNC	same	This is now a function, not a CALL routine.
XCOMGET		no equivalent	This is no longer necessary.
XEBUG	SAS_XEBUG	same	
XEXIT	SAS_XEXIT	same	
XFFILE	SAS_XFFILE	same	This is now a function, not a CALL routine.
XFNAME	SAS_XFNAME	same	This is now a function, not a CALL routine.
XFSHOW	SAS_XFSHOW	same	
XFXFN		no equivalent	
XFXIC	SAS_XFXIC	same	This is now a function, not a CALL routine.
XFXIN	SAS_XFXIN	same	This is now a function, not a CALL routine.
XGETBOP	SAS_XOPTBGT	new routine	You now specify the actual name of the option, not a numeric code for it. More options can be tested.
XGETIOP	SAS_XOPTIGT	new routine	See the description of SAS_XOPTBGT.
XINIT	UWPRCC, UWPRCF or UWPRCP	new routine	
XLOG1	SAS_XPSLOG1	some differences	All printing routines have new substitution characters.
XLOG2	SAS_XPSLOG2	some differences	All printing routines have new substitution characters.
XLOG3	SAS_XPSLOG3	some differences	All printing routines have new substitution characters.
XLOG4	SAS_XPSLOG4	some differences	All printing routines have new substitution characters.
XLOG5	SAS_XPSLOG5	some differences	All printing routines have new substitution characters.
XLOGA	SAS_XPSLOG	same	

(continued)

Table 17.3 *(continued)*

Version 5 X Routine	Version 6 SAS_X Routine	Comparison	Notes
XMEMDBL		no equivalent	Version 6 no longer provides this routine because doubling memory causes memory fragmentation. It is better to allocate the memory you need, use it, and free it after you have finished using it.
XMEMDEF		no equivalent	Version 6 no longer provides this routine because it is safer to check for a NULL pointer after each memory allocation instead of checking a deficit value after multiple allocations.
XMEMEX	SAS_XMEMEX	same	
XMEMFRE	SAS_XMEMFRE	same	This is now a function, not a CALL routine.
XMEMGET	SAS_XMEMGET	same	
XMEMMSG	SAS_XMEMZER and SAS_XPSLOG	different routines	You can no longer report how much memory was unavailable, but you can use SAS_XPSLOG to report how much was requested when the allocation failed.
XMEMZER	SAS_XMEMZER	same	
XMPOOL	SAS_XMPOOLC or SAS_XMALLOC	new routines	SAS_XMPOOLC enables you to create a pool. SAS_XMALLOC lets you use memory from the allocated pool. XMPOOL combined these capabilities.
XMPOOLF	SAS_XMPOOLD	new routine	SAS_XMPOOLD is similar to XMPOOLF, but you now can free portions of a pool by using SAS_XMFREE. SAS_XMPOOLD is a function, not a CALL routine.
XMPOOLS	SAS_XMPOOLC	new routine	SAS_XMPOOLC only creates a pool for later use; you must also call SAS_XMALLOC to allocate memory for a specific use. With SAS_XMPOOLC, you can specify initial and overflow allocations, not just a single size of memory.
XOANY	SAS_XOINFO	new routine	Call SAS_XOINFO with the XO_ANY flag.
XOCLOSE	SAS_XOCLOSE	some differences	You must specify with SAS_XOCLOSE whether to save the data set or not. This is now a function, not a CALL routine.
XOCLOSN		no equivalent	

(continued)

Table 17.3 *(continued)*

Version 5 X Routine	Version 6 SAS_X Routine	Comparison	Notes
XOGET	SAS_XBYGET or SAS_XOGETN	new routine	Use SAS_XBYGET for all procedures that permit a BY statement. Use SAS_XOGETN only when a BY statement is not permitted. SAS_XOGETN has an additional argument.
XOGLAB	SAS_XOINFO	new routine	Call SAS_XOINFO with the XO_LABEL flag.
XOGPTR	SAS_XOGETN with SAS_XOINFO	new routine	Call SAS_XOINFO with the CUROBS code to obtain the current observation pointer. Then call SAS_XOGETN with this pointer.
XOMISS	SAS_XVMISS	new routine	You can now initialize the output buffer.
XONOT	SAS_XONOTE	new routine	SAS_XONOTE returns a record ID pointer, not an observation number.
XOOBSN	SAS_XOINFO	new routine	Call SAS_XOINFO with the SAS_XONOBS flag.
XOOPEN	SAS_XOOPEN	some differences	The order of the arguments is different. The XOPNSTR structure is different, and the return codes have changed. This is now a function, not a CALL routine.
XOPLAB	SAS_XOINFO	new routine	Call SAS_XOINFO with the XO_SLABEL or XO_STYPE flag.
XOPNT	SAS_XOPNT	some differences	SAS_XOPNT expects a record ID pointer, not an observation number. This is now a function, not a CALL routine.
XOPPTR	SAS_XOINFO	new routine	Call SAS_XOINFO with the XO_CUROBS flag.
XORPTR	SAS_XOINFO	new routine	Call SAS_XOINFO with the XO_CUROBS flag.
XORWND	SAS_XBYRWND	new routine	The rewind type code for the beginning of the file has changed. SAS_XBYRWND is now a function, not a CALL routine.
XPAGE	SAS_XPAGE	same	Although this routine is still available, Version 6 provides many new printing capabilities with new routines. See "SAS_XP Routines: Printing" in Chapter 22 for more information.

(continued)

Table 17.3 *(continued)*

Version 5 X Routine	Version 6 SAS_X Routine	Comparison	Notes
XPC	SAS_XPC	same	Although this routine is still available, Version 6 provides many new printing capabilities with new routines. See "SAS_XP Routines: Printing" in Chapter 22 for more information.
XPCR		no equivalent	
XPCS	SAS_XPCS	same	Although this routine is still available, Version 6 provides many new printing capabilities with new routines. See "SAS_XP Routines: Printing" in Chapter 22 for more information.
XPF	SAS_XPF	same	Although this routine is still available, Version 6 provides many new printing capabilities with new routines. See "SAS_XP Routines: Printing" in Chapter 22 for more information.
XPFS	SAS_XPFS	same	Although this routine is still available, Version 6 provides many new printing capabilities with new routines. See "SAS_XP Routines: Printing" in Chapter 22 for more information.
XPF12		no equivalent	
XPI	SAS_XPI	same	Although this routine is still available, Version 6 provides many new printing capabilities with new routines. See "SAS_XP Routines: Printing" in Chapter 22 for more information.
XPIS	SAS_XPIS	same	Although this routine is still available, Version 6 provides many new printing capabilities with new routines. See "SAS_XP Routines: Printing" in Chapter 22 for more information.
XPI12		no equivalent	
XPSKIP	SAS_XPSKIP	same	Although this routine is still available, Version 6 provides many new printing capabilities with new routines. See "SAS_XP Routines: Printing" in Chapter 22 for more information.

(continued)

Table 17.3 *(continued)*

Version 5 X Routine	Version 6 SAS_X Routine	Comparison	Notes
XRCLOSE	SAS_XCCLOSE	new routine	Codes for the disposition of the file after closing have changed. This is now a function, not a CALL routine.
XRGET	SAS_XCREAD	new routine	You no longer specify the length of the record storage area, only the length of the data to read. This is now a function, not a CALL routine.
XRGETL		no equivalent	
XRNOT	SAS_XCNOTE	new routine	You now receive a position, not a record number, to point to a record.
XRNOTP	SAS_XCNOTE	new routine	You now receive a position number, not a block of data, to point to the record. You no longer need to use different routines for variable-length versus fixed-length files. This is now a function, not a CALL routine.
XRNOTPN	SAS_XCNOTE	new routine	The calling syntax is slightly different. You no longer need to use different routines for variable-length versus fixed-length files. This is now a function, not a CALL routine.
XROPEN	SAS_XCOPEN	new routine	The calling sequence is different. You no longer use the XOPNSTR for utility files. You complete the XCOPNSTR before calling SAS_XCOPEN. This is now a function, not a CALL routine.
XRPNT	SAS_XCPOINT	new routine	You now use the position, not a record number, to point to a record. This is now a function, not a CALL routine.
XRPNTP	SAS_XCPOINT	new routine	You now use the position, not a block of data, to point to a record. You no longer need to use different routines for variable-length versus fixed-length files. This is now a function, not a CALL routine.
XRPUT	SAS_XCWRITE	new routine	This is now a function, not a CALL routine. SAS_XCWRITE works like the Version 5 XRPUT routine.
XSPARSC		no equivalent	
XSPARSE	SAS_XSPARSE	some differences	You must provide additional arguments. This is now a function, not a CALL routine.
XTOKE		no equivalent	This step is no longer necessary.

(continued)

Table 17.3 *(continued)*

Version 5 X Routine	Version 6 SAS_X Routine	Comparison	Notes
XVCOUNT	SAS_XOINFO	new routine	Call SAS_XOINFO with the XO_NLOBS flag.
XVDFLST	SAS_XVDFLST	some differences	You now omit lists from the default list by specifying the STLIST element for the list. The calling order for SAS_XVDFLST is no longer significant. This is now a function, not a CALL routine.
XVFIND	SAS_XVFIND	some differences	SAS_XVFIND now indicates if a variable is in a data set, but it does not check for variables in a specific list. Use the variable number returned by SAS_XVFIND to compare to the variable numbers in lists to determine if the variable is included in a list.
XVGET	SAS_XVGET	some differences	You now specify in the call to SAS_XVGETD whether the move is for absolute addresses or offsets from a base address. When the move is for absolute addresses, you specify NULL for the base address in the call to SAS_XVGET. This type of call to SAS_XVGET is equivalent to the Version 5 call to XVGET. This is now a function, not a CALL routine.
XVGETD	SAS_XVGETD	some differences	You now use the variable number from the input data set, not the number of the variable in any specific list. In addition, you can specify either offset moves or specific addresses for moving data. SAS_XVGETD combines the capabilities of XVGETD and XVGETDO. In Version 6, you must call SAS_XVGETE at the end of the calls to SAS_XVGETD. This is now a function, not a CALL routine.
XVGETDO	SAS_XVGETD	new routine	See the description for XVGETD.
XVGETI	SAS_XVGETI	same	This is now a function, not a CALL routine.
XVGETT	SAS_XVGETT	same	This is now a function, not a CALL routine.
XVGET1C	SAS_XVGET1C	some differences	This routine no longer uses list numbers, only variable numbers from the input data set. This is now a function, not a CALL routine.

(continued)

Table 17.3 *(continued)*

Version 5 X Routine	Version 6 SAS_X Routine	Comparison	Notes
XVGET1F	SAS_XVGET1F	some differences	This routine no longer uses list numbers, only variable numbers from the input data set. This is now a function, not a CALL routine.
XVGET1V	SAS_XVGET1V	some differences	This routine no longer uses list numbers, only variable numbers from the input data set. This is now a function, not a CALL routine.
XVGET2	SAS_XVGET	new routine	SAS_XVGET now works like XVGET2 did in Version 5. Use SAS_XVGETD to specify the offsets and use SAS_XVGET with a base pointer. This is now a function, not a CALL routine.
XVLIST		no equivalent	This step is no longer necessary.
XVNAME	SAS_XVNAME	some differences	You now use the variable number from the input data set, not the number of the variable in any specific list. This is now a function, not a CALL routine.
XVNAMEI	SAS_XVNAMEI	same	This is now a function, not a CALL routine.
XVNAMEL		no equivalent	Use built-in functions.
XVPUT	SAS_XVPUT	some differences	SAS_XVPUT no longer writes an observation. You must use SAS_XOADD to actually write the observation. This is now a function, not a CALL routine.
XVPUTD	SAS_XVPUTD	same	This is now a function, not a CALL routine.
XVPUTE	SAS_XVPUTE	same	This is now a function, not a CALL routine.
XVPUTI	SAS_XVPUTI	same	This is now a function, not a CALL routine.
XVPUTT	SAS_XVPUTT	same	This is now a function, not a CALL routine.
XVPUT2	SAS_XVPUT	same	Use SAS_XVPUTD to specify the offsets and use SAS_XVPUT with a base pointer. This is now a function, not a CALL routine.

Option Handling

In Version 5, you used the XGETBOP routine to obtain the settings of bit arguments, SAS system options that have a value of *option*/NO*option*. You used XGETIOP to obtain the value of a numeric parameter option, a SAS system option assigned a numeric value. For both of these routines, you specified a numeric code to indicate which option to test. In Version 6, you specify the option name itself. Table 17.4 lists the Version 5 option routines and numbers, and the corresponding Version 6 routines and names. Note that there are more options, including character options, that you can test in Version 6. Refer to the descriptions of the SAS_XOPT routines in Chapter 22.

Table 17.4 *Testing System Options*

Version 5 Routine	Version 5 Option Number	Version 6 Routine	Version 6 Option Name
XGETBOP	33005	SAS_XOPTBGT	MACRO
	34001		CENTER
	34011		OVP
	34050		DSNFERR
	38002		LABEL
	38007		FMTERR
	47001		DMS
XGETIOP	34001	SAS_XOPTIGT	LINESIZE
	34003		PAGESIZE
	36003		FIRSTOBS
	36005		OBS

The following examples illustrate how to use each routine in Version 5 and Version 6:

Version 5 method

```
DCL LINESIZE FIXED BIN(31),
    USER CHAR(8),
    FMTERR BIT(1);
...
LINESIZE = XGETIOP(34001);
FMTERR = XGETBOP(38007);
```

Version 6 method

```
DCL (LINESIZE,RC) FIXED BIN(31),
    TEMP PTR,
    USER CHAR(8),
    FMTERR BIT(1);
...
LINESIZE = SAS_XOPTIGT('LINESIZE',
                       ADDR(RC));
TEMP = SAS_XOPTCGT('USER',
                   ADDR(RC));
CALL SAS_ZMOVEI(TEMP,
                ADDR(USER),8);
FMTERR = (SAS_XOPTBGT('FMTERR',
          ADDR(RC)) = 1);
```

Memory Management

In Version 5, the pool-handling routines, XMPOOL and XMPOOLS, were used to allocate a pool of memory and also to use parts of the pool. You could specify the size of the pool as well as the amount needed for the current allocation in a call to XMPOOLS, but the call to XMPOOL used a default host-dependent pool size. Pools of memory were automatically set to 0 when they were allocated. When you were finished with the pool, you could free the entire pool by using XMPOOLF.

In Version 6, you initialize the pool and indicate its size by calling SAS_XMPOOLC. You can either specify the size of the pool or indicate that the system should use a host-dependent default size for the pool, by using the XM_ISA and XM_OSA symbols. Version 6 allows you to specify two allocation sizes: the size of initial allocation and the amount to add to that allocation each time after the first. If you want allocated memory set to 0s, you must specify the XM_ZERO flag in the call to SAS_XMPOOLC. To use portions of the pool, you call SAS_XMALLOC. To delete the entire pool use SAS_XMPOOLD. You can now free portions of the pool by calling SAS_XMFREE.

This set of examples illustrates how to create a pool of memory in Version 5 and Version 6:

Version 5 method

```
PTR = XMPOOL(POOLHEAD,NEED);
PTR = XMPOOLS(POOLHEAD,NEED,
              NODESIZE);
```

Version 6 method

```
PTR = SAS_XMPOOLC(ISA_SIZE,
                  OSA_SIZE,FLAGS);
```

The following set of examples shows how to create and use a pool of memory in Version 5 and Version 6. Both examples use the system default for the pool size, and both examples free the entire pool at the end.

Version 5 method

```
DCL (P(3),PID) PTR;
...
PID = $NULL();
P(1) =  XMPOOL(PID,100);
P(2) =  XMPOOL(PID,200);
P(3) =  XMPOOL(PID,300);
...
CALL XMPOOLF(PID);
```

Version 6 method

```
DCL (PID,P(3)) PTR;
...
PID = SAS_XMPOOLC(XM_ISA,
                  XM_OSA,XM_ZERO);
P(1) = SAS_XMALLOC(PID,100,0);
P(2) = SAS_XMALLOC(PID,200,0);
P(3) = SAS_XMALLOC(PID,300,0);
...
CALL XMPOOLD(PID);
```

SAS Data Set I/O Routines

The following sections discuss differences in Version 5 and Version 6 in methods for opening and closing SAS data sets, obtaining information on data sets, setting the type and label for an output data set, reading observations, rewinding data sets, and directly accessing a SAS data set.

Opening and Closing SAS Data Sets

The Version 5 XOOPEN and Version 6 SAS_XOOPEN routines open a SAS data set in the middle of a procedure. In both versions of the system, most SAS data sets are opened by the @DS or @DSDFLT semantic actions during the parsing phase. There may be cases, however, when your procedure needs to open a data set within the procedure. The process for opening a data set is the same, but the structure used to pass information about the data set has changed and the syntax for SAS_XOOPEN routine is different. It is now a function, not a CALL routine, and you now pass the address of a fileid, not the fileid itself. These examples illustrate the changes:

Version 5 method

```
CALL XOOPEN(PARMPTR,FILEID,RC);
```

Version 6 method

```
RC = SAS_XOOPEN(PARMPTR,
                ADDR(FILEID));
```

The structures used to pass information about a SAS data set are different. The most pertinent fields still exist in Version 6. Fill those in as appropriate before calling SAS_XOOPEN. The following examples illustrate how to prepare the structure needed to open a data set:

Version 5 method

```
%INCLUDE XOPNSTR;
XOPNLIB=' ';
XOPNMEM='TEST';
XOPNMODE='INPUT'
XOPNEXT=' ';
XOPNLREC=0;
XOPNFVU=' ';
XOPNREAD=' ';
XOPNPROT=' ';
CALL XOOPEN(ADDR(XOPNSTR),
            FILEID,RC);
```

Version 6 method

```
DCL  1 XOPNSTR,
%INCLUDE XOPNSTR;
XOPNSTR.LIBNAME=' ';
XOPNSTR.MEMNAME='TEST';
XOPNSTR.OPNMODE=XO_INPUT;
XOPNSTR.MAXRC=0;
RC=SAS_XOOPEN(ADDR(FILEID),
              ADDR(XOPNSTR));
```

In Version 5, the XOCLOSE routine simply closed the data set. In Version 6, you specify an additional argument to indicate what to do with the closed data set. To use SAS_XOCLOSE so that it mimics the action of Version 5, use a 0 for disposition, as shown in this set of examples.

Version 5 method

```
CALL XOCLOSE(FID);
```

Version 6 method

```
CALL SAS_XOCLOSE(FID,0);
```

Obtaining and Setting Information on Data Sets

To obtain information on SAS data sets in Version 5, you used several routines and the LABELSTR structure. The XOANY routine determined if the data set had any variables or observations to process. The Version 5 XOGLAB routine obtained global information about the SAS data set and copied it into the LABELSTR structure. The XOPLAB routine used the same structure to update the information in the data set. The Version 5 XOOBSN routine returned the number of observations from the input data set.

In Version 6, you obtain information and set the data set type and label by using one routine, SAS_XOINFO, with different arguments.

Checking Observations

To determine if the data set has observations and variables, call SAS_XOINFO with the XO_ANY value. The return values are the same as those returned in Version 5.

1 indicates the data set has observations.

0 indicates the data set has no observations but it does have variables.

−1 indicates there are no variables.

The following set of examples illustrates the two methods of checking the data set:

Version 5 method

```
IF (XOANY(FILEID) = 0) THEN DO;
   CALL XLOG1('NOTE: No observations
               in data set.');
   .....
   END;
```

Version 6 method

```
DCL I FIXED BIN(31);
...
CALL SAS_XOINFO(FILEID,XO_ANY,
                ADDR(I));
IF (I = 0) THEN DO;
   CALL SAS_XPSLOG1('%1zNo observations
                     in data set.');
   .....
   END;
```

In Version 5, the XOOBSN routine returned the number of observations in the input data set. If the data set was in tape format, XOOBSN returned −1. In Version 6, you call SAS_XOINFO with the XO_NLOBS to determine the number of observations. As in Version 5, SAS_XOINFO returns a −1 for sequential SAS data sets.

Note: In Version 6, the number of observations returned by SAS_XOINFO should be treated as the *maximum* number of observations, not as an exact number. If the data set is not subsetted by the FIRSTOBS= or OBS= system option and no WHERE processing is in effect, the number returned indicates the actual number of observations. If the data set is subsetted by the FIRSTOBS= or OBS= options, the number returned indicates the number of selected observations, but any observations marked for deletion are also included in the count. Under no circumstances does the number of observations reflect subsetting by WHERE processing. If your application must know the number of observations, read through the data set counting the observations and then rewind the data set for processing, using SAS_XBYRWND with the 'F' code.

The following set of examples illustrates how to obtain the number of observations in the data set:

Version 5 method

```
N = XOOBSN(FILEID);
```

Version 6 method

```
CALL XOINFO(FILEID,XO_NOBS,
            ADDR(N));
```

Reading and Setting the Data Set Type and Label

The Version 6 equivalent to the XOGLAB routine is to use SAS_XOINFO with the XO_LIB, XO_MEM, XO_TYPE, and XO_LABEL codes. The following set of examples shows how to use the Version 5 and Version 6 method for obtaining data set information.

Note: The Version 6 method in this example continues to use a LABELSTR structure to store the information about the data set, but this structure is no longer included with the system. You can either create your own structure for storing this data, or simply store it in separate variables.

Version 5 method

```
CALL SAS_XOGLAB(FILEID,LABELPTR);
```

Version 6 method

```
CALL SAS_XOINFO(FILEID,XO_LIB,
          ADDR(LABELPTR->LABLIB));
CALL SAS_XOINFO(FILEID,XO_MEM,
          ADDR(LABELPTR->LABMEM));
CALL SAS_XOINFO(FILEID,XO_TYPE,
          ADDR(LABELPTR->LABTYP));
CALL SAS_XOINFO(FILEID,XO_LABEL,
          ADDR(LABELPTR->LABLAB));
```

To set the data set type and label in Version 6, you use SAS_XOINFO with the XO_STYPE and XO_SLABEL codes. This parallels the Version 5 call to SAS_XOPLAB, as illustrated in this set of examples. Note that you cannot change the libref or member name of the data set.

Version 5 method

```
LABLAB='NUMERIC VARIABLE ONLY';
LABTYP='CORR';
CALL SAS_XOPLAB(OUTFILEID,LABELPTR);
```

Version 6 method

```
LABELPTR->LABLAB=
  'NUMERIC VARIABLE ONLY';
LABELPTR->LABTYP='CORR';
CALL SAS_XOINFO(FILEID,XO_STYPE,
          ADDR(LABELPTR->LABTYP));
CALL SAS_XOINFO(FILEID,XO_SLABEL,
          ADDR(LABELPTR->LABLAB));
```

Reading Observations from Data Sets

The changes to reading observations are primarily in the following areas:

- BY-group processing (XOGET and XORWND)
- accessing the observation buffer (XOGPTR and XOPPTR)
- direct access methods (XONOT and XOPNT).

Each of these areas is discussed in detail in the following sections.

BY-Group Processing

The Version 5 XOGET routine read in an observation, returning a 0 if the observation was read, and a 1 if not. In Version 6, the SAS_XBYGET routine provides the same capability as SAS_XOGET. Enclose the SAS_XBYGET call in a

SAS_XBYNEXT loop. If the user does not specify a BY statement, the loop processes the entire data set instead of looking for BY groups.

The Version 5 XORWND rewinds a data set to the beginning of the BY group or the beginning of the data set depending on the second argument being '1' or 'B'. In Version 6, use the SAS_XBYRWND routine with an argument of 'B' to rewind to the beginning of the BY group and 'F' to rewind to the beginning of the data set.

Note: Do not use SAS_XOPNT with the XO_BOD flag to rewind to the first observation because the SAS_XOPNT routine ignores the FIRSTOBS= option. When you call SAS_XBYRWND with the 'F' argument, it rewinds to the first observation selected for processing.

Version 5 method

```
   /* to first observation */
CALL XORWND(FILEID,'1');
   /* to start of by-group */
CALL XORWND(FILEID,'B');
```

Version 6 method

```
   /* to first observation */
CALL SAS_XBYRWND(FILEID,'F');
   /* to start of by-group */
CALL SAS_XBYRWND(FILEID,'B');
```

Accessing the Observation Buffer

In Version 5, you used the XOGPTR routine to read observations when no BY-group processing was permitted. The XOGPTR routine returned a pointer to the observation buffer, which could be used to access the observation buffer, but was not needed if you used the XVGET routines to scatter read the observation. The corresponding routine in Version 6, SAS_XOGETN, reads an observation without concern for BY-group processing, but it does not return the pointer to the observation buffer. In most cases, you use the SAS_XVGET routines to scatter read so you do not need the pointer to the observation buffer. To access the observation buffer pointer, you must call the SAS_XOINFO routine with the XO_CUROBS flag. Note that this buffer pointer can change, so you must call SAS_XOINFO after every SAS_XOGETN call.

Version 5 method

```
NULO = $NULL();
P = XOGPTR(FILEID);
DO WHILE(P ¬= NULO);
      /* operate on P */
   P = XOGPTR(FILEID);
   END;
```

Version 6 method

```
NULO = SAS_NUL();
CALL SAS_XOGETN(FILEID,ADDR(P));
DO WHILE(P ¬= NULO);
      /* operate on P */
   CALL SAS_XOGETN(FILEID,ADDR(P));
   END;
```

In Version 5, to access the observation buffer when you were processing BY groups, you called XOGET and then called XOPPTR. In Version 6, call XBYGET and then call SAS_XOINFO with the XO_CUROBS code. However, do not call SAS_XOINFO with XO_CUROBS if SAS_XBYGET has returned a 1. This is because the buffer pointer may point to the first observation in the next BY group.

Version 5 method

```
DO WHILE(XBYNEXT(FILEID) > 0);
   DO WHILE(XOGET(FILEID) = 0);
      P = XOPPTR(FILEID);
      /* operate on P */
      END;
   END;
```

Version 6 method

```
DO WHILE(SAS_XBYNEXT(FILEID) > 0);
   DO WHILE(SAS_XBYGET(FILEID) = 0);
      CALL SAS_XOINFO(FILEID,XO_CUROBS,
                      ADDR(P));
         /* operate on P */
      END;
   END;
```

Direct Access

The Version 5 XONOT and XOPNT routines provided the position of the current observation and saved this position for pointing back to the observation at some later point in processing. The position information was stored in an integer, which was, in effect, the observation number. In Version 6, this same process exists, but the position information is stored instead in a record ID, also known as a *rid*. The new file concepts available with Version 6, such as compressed data sets, indexes, and the many engines, make the record ID a more useful form of identifier than the observation number. To call SAS_XONOTE, the Version 6 equivalent to the Version 5 XONOT, you must first determine the size of the record ID for the data set your procedure is processing and then allocate space to store the rid. Use SAS_XOINFO with the XO_LRID flag to determine the size of the rid. Pass a pointer to the allocated space to SAS_XONOTE, which fills in the record ID. When you want to point back to the observation, call SAS_XOPNT with this rid. There are several special rids for convenience, including XO_BOD, which points to the beginning of data.

Version 5 method

```
DCL OBSNUM FIXED BIN(31);
...
   /* SET OBSNUM */
CALL XONOT(FILEID,OBSNUM);
...
   /* POINT BACK TO THE OBS */
CALL XOPNT(FILEID,OBSNUM);
```

Version 6 method

```
DCL RID PTR;
DCL RIDLEN FIXED BIN(31);
...
CALL SAS_XOINFO(FILEID,XO_RIDLEN,
                ADDR(RIDLEN));
RID = SAS_XMEMGET(RIDLEN);
...
   /* SET DATA IN RID */
CALL SAS_XONOTE(FILEID,RID);
...
   /* POINT BACK TO THE OBS */
CALL SAS_XOPNT(FILEID,RID);
```

Printing Routines

If you are converting a Version 5 procedure in PL/I to Release 6.07 or later, you can choose which set of printing routines to use. The printing routines provided with Version 5 are still available under Release 6.07, but new printing routines are also available. If you choose to use the routines that are similar to Version 5, there are still some differences you need to note. These include the following:

- Substitution characters in strings function differently.
- You no longer access the LINESTR structure to manipulate the printing environment. You now use several new SAS_XP routines.
- Under most operating systems, you can use a single routine to print to the log.

Substitution Characters

The substitution characters in printing routines are different in Version 6. The # substitution in Version 5 has been replaced by C-like directives in Version 6. In Version 5, all items substituted for # were character strings. In Version 6, you can simply change the # substitution character to %s for character strings, but you must use new substitution characters to print integers or floating-point values.

An important difference between Version 5 and Version 6 is that you must use the correct number of arguments in Version 6. Version 5 XLOG routines allowed you to have multiple arguments where there was no # substitution. Additional arguments were concatenated with the first string. The Version 6 routines do not concatenate extra arguments for which there are no substitution characters. If you do not specify enough % substitution characters in the first argument, the extra arguments are ignored.

The following examples illustrate corresponding Version 5 and Version 6 calls to print to the log:

Version 5 method

```
CALL XLOG1('NO SUBSTITUTIONS,
          1 STRING');
CALL XLOG2('SUBSTITUTION: #.',
          'INSERTED HERE');
CALL XLOG2('INTEGER VALUE: #.',
          XPI12(INT_VALUE));
CALL XLOG2('FLOAT VALUE: #.',
          XPF12(FLOAT_VALUE,2));
```

Version 6 method

```
CALL SAS_XPSLOG('NO SUBSTITUTIONS,
                 1 STRING');
CALL SAS_XPSLOG('SUBSTITUTION: %s.',
                 'INSERTED HERE');
CALL SAS_XPSLOG('INTEGER VALUE: %d.',
                 INT_VALUE);
CALL SAS_XPSLOG('FLOAT VALUE: %f.',
                 FLOAT_VALUE);
```

Before attempting to use the new style of formatting characters, you need to read "SAS_XP Routines: Printing" in Chapter 22 for complete information.

Working in the Printing Environment

In Version 5, you used the LINESTR structure to determine the settings of some system options and to change default behavior of the printing routines. Table 17.5 summarizes the Version 5 use of the LINESTR and the corresponding method to use in Version 6.

Table 17.5 *Comparison of Version 5 and Version 6 Printing Flags*

Check or Set This Value	Field in the Version 5 LINESTR Structure	Version 6 Technique
the number of lines to print before the BY line.	LINEBYLINEPOS	Call SAS_XPOPTGT or SAS_XPOPTST with the XPBYLINPOS flag.
the value of the CENTER SAS system option.	LINECENTER	Call SAS_XPOPTGT with the XPCENTER flag.
the headings to be printed on each page of output.	LINEHPTR	Use SAS_XPHDR to define up to ten lines of headers.
the starting column in the output buffer where the string will be placed.	LINEINDENT	Call SAS_XPOPTGT or SAS_XPOPTST with the XPLMARGIN.
the line number of the line being created.	LINENUM	Call SAS_XPOPTGT with the XPLINENUM flag.
the flag to indicate when a new page should be started.	LINEONE	Call SAS_XPOPTGT with the XPLINEONE flag.
the flag that suppresses footnotes.	LINEOPT(1)	Call SAS_XPOPTGT or SAS_XPOPTST with the XPNOFOOTS flag.
the flag that suppresses page numbers in the heading.	LINEOPT(2)	Call SAS_XPOPTGT or SAS_XPOPTST with the XPNOPAGE flag.
the flag that suppresses the date in the heading.	LINEOPT(3)	Call SAS_XPOPTGT or SAS_XPOPTST with the XPNODATE flag.
the flag that suppresses the default BY line.	LINEOPT(4)	There is no equivalent. The NOBYLINE SAS system option prevents this capability from being available.
the flag that suppresses titles.	LINEOPT(5)	Call SAS_XPOPTGT or SAS_XPOPTST with the XPNOTITLE flag.
the flag that suppresses the blank line following header lines.	LINEOPT(6)	Call SAS_XPOPTGT or SAS_XPOPTST with the XPNOBLANK flag.
the line to be printed.	LINEOUT	This is the output buffer. You place character strings in this buffer with the SAS_XPS routine.

(continued)

Table 17.5 *(continued)*

Check or Set This Value	Field in the Version 5 LINESTR Structure	Version 6 Technique
the page number of the page being printed.	LINEPAGENUM	Call SAS_XPOPTGT with the XPPAGENUM flag.
the value of the PAGESIZE= option.	LINEPGSIZE	Call SAS_XPOPTGT with the XPPAGESIZE flag.
the last column number used in the output buffer.	LINEPOS	Call SAS_XPOPTGT or SAS_XPOPTST with the XPCOLNUM flag.
the maximum number of bytes in the output buffer that you can use.	LINESIZE	Call SAS_XPOPTGT with the XPLINESIZE flag.
the number of lines left to print in the body of the report. The number returned is actually one less than the number left.	LINESLEFT	Call SAS_XPOPTGT with the XPLEFT flag.
the highlight or color of a line.	LINETYPE	Use the %h substitution character in formatting the string.

Log Processing

In Version 5, log printing was handled by one of the XLOG1 through XLOG5 or XLOGA routines, depending on the number of arguments. XLOGA was used when no substitution was to take place. In Version 6 on most operating systems, you can use SAS_XPSLOG to replace all XLOGA and XLOG1 through XLOG5 calls. If you are writing your procedure under VMS or if it may ever be ported to VMS, you need to use the SAS_XPSLOG1 routine for XLOGA, and SAS_XPSLOG1 through SAS_XPSLOG5 for the XLOG1 through XLOG5 routines because the PL/I compiler under VMS needs to have the exact number of arguments declared for a routine. The SAS_XPSLOG1 though SAS_XPSLOG5 routines are provided for this specific requirement. The Version 6 implementation under VMS also provides SAS_XPSLOG6 through SAS_XPSLOG10 for added functionality.

Version 5 method

```
CALL XLOG1('NO SUBSTITUTIONS,
           1 STRING');
CALL XLOG2('NO SUBSTITUTIONS.',
           '2 STRINGS');
CALL XLOG2('INTEGER VALUE: #.',
           XPI12(INT_VALUE));
CALL XLOG2('FLOAT VALUE: #.',
           XPF12(FLOAT_VALUE,2));
```

Version 6 method

```
CALL SAS_XPSLOG1('NO SUBSTITUTIONS,
                  1 STRING');
CALL SAS_XPSLOG2('ONE ARGUMENT TO %s',
                 'SUBSTITUTE');
CALL SAS_XPSLOG2('INTEGER VALUE: %d.',
                  INT_VALUE);
CALL SAS_XPSLOG2('FLOAT VALUE: %f.',
                  FLOAT_VALUE);
```

Note: In Version 6, you cannot specify a constant for a numeric value in a call to one of the SAS_XPSLOG routines. In the preceding examples, you must specify a variable, such as INT_VALUE or FLOAT_VALUE, for the numeric arguments to be substituted into the string.

Utility File I/O

Version 6 handles utility file I/O differently in several areas:

- The prefix for the names of the routines has changed from XR to SAS_XC, and the names of the routines have changed somewhat.
- You now use the same routines to handle both fixed-length and variable-length files.
- When you note a location in a file, it is the location of the *next* record to be read, not the one you are currently processing.
- The XRGETL routine has no equivalent in Version 6.
- The SAS_XC routines use a special data structure, XCOPNSTR, instead of using the same structure as the SAS_XO routines as in Version 5.

These points are discussed in more detail in the following sections.

Noting a Position in Fixed-length and Variable-length Records

In Version 5, you used one set of routines, XRNOT and XRPNT, to handle fixed-length records and a different set of routines, XRNOTP, XRNOTPN, and XRPNTP, for variable-length record files. Both kinds of files were read and written by the XRGET and XRPUT routines. Notes for fixed-length files were actually record numbers, which were determined by XRNOT, and used by XRPNT to point to particular records. For variable-length record files, XRNOTP set a note field that was used by XRPNTP to point to the record. XRNOTPN noted the position of the next record.

In Version 6, you use one set of routines for all types of records. SAS_XCNOTE notes the position of a record, but in Version 6 the record noted is always the next record to be read. To note the position of the current record, you must call SAS_XCNOTE before calling SAS_XCREAD. The Version 6 implementation of SAS_XCNOTE is equivalent to the Version 5 XRNOTP routine.

Version 6 no longer provides the ability to query the length of a variable-length record before reading it. In Version 5, this was accomplished by calling XRGETL to get the length before calling XRGET to read the record.

Data Structure for Opening Files

In Version 5, the XR routines used the XOPNSTR structure for opening files. In Version 6, the SAS_XC routines use the XCOPNSTR structure. The following set of examples illustrates the methods for completing these data structures before opening a utility data set:

Version 5 method

```
DCL STUFF CHAR(20);
DCL READ_STUFF CHAR(20);
DCL BYTES_READ FIXED BIN(31);
...
STUFF='STUFF TO WRITE OUT';
XOPNLIB=' ';
XOPNMEM='XXX';
XOPNMODE='OUTPUT';
XOPNEXT='UTILITY';
XOPNLREC=20;
XOPNFVU='F';
XOPNREAD=' ';
XOPNPROT=' ';
CALL XROPEN(ADDR(XOPNSTR),
            FILEID,RC);
CALL XRPUT(FILEID,ADDR(STUFF),20);

   /* NOTE OCCURS AFTER WRITE */
CALL XRNOTE(FILEID,STUFF_NOTE);
STUFF='NEW RECORD';
CALL XRPUT(FILEID,ADDR(STUFF),20);
CALL XRPNT(FILEID,STUFF_NOTE);
CALL XRGET(FILEID,ADDR(READ_STUFF),
           20,BYTES_READ);
```

Version 6 method

```
DCL STUFF CHAR(20);
DCL READ_STUFF CHAR(20);
...
STUFF='STUFF TO WRITE OUT';
FN.LIBNAME=' ';
FN.MEMNAME='XXX';
FN.MEMTYPE='UTILITY';
FN.LIB_HANDLE = NULO;
MAX_RC = 0;
MODE = XC_OUTPUT;
RC = SAS_XCOPEN(ADDR(XCOPNSTR),
                ADDR(FILEID));
   /* NOTE OCCURS BEFORE THE WRITE */
RC = SAS_XCNOTE(FILEID,ADDR(NOTE));
RC = SAS_XCWRITE(FILEID,
                 ADDR(STUFF),20);
STUFF='NEW RECORD';
RC = SAS_XCWRITE(FILEID,
                 ADDR(STUFF),20);
RC = SAS_XCPOINT(FILEID,NOTE);
RC = SAS_XCREAD(FILEID,
                ADDR(READ_STUFF),20);
```

Parsing Routines

In Version 5, the XSPARSE routine was called and given the grammar pointer. XSPARSC was equivalent, except that an entry variable was passed as the second argument to indicate the custom semantic action routine.

In Version 6, the SAS_XSPARSE routine is passed the grammar function pointer, as well as the address of the PROC structure and a NULL pointer. At this time, custom semantic actions are not supported so the XSPARSC routine has no Version 6 equivalent.

The following set of examples illustrates the old and new methods of parsing:

Version 5 method

```
   /* GRAMMAR FUNCTION */
DCL ABCG ENTRY RETURNS(PTR);
CALL XSPARSE(ABCG());
```

Version 6 method

```
   /* GRAMMAR FUNCTION */
DCL ABCG ENTRY RETURNS(PTR);
DCL NUL0 PTR;
NUL0 = SAS_NUL();
CALL SAS_XSPARSE(ABCG(),NUL0,
                 ADDR(PROC));
```

Variable Processing

Variable processing has changed in the following areas:

- list processing
- scatter reading
- gather writing.

These changes are explained in detail in the following sections.

List Processing

In Version 5, many of the XV routines referred to variables by their number in a specific list. For example, if the procedure permitted a VAR list (or created a default one when none was specified), the number associated with the VAR list as well as the number of the variable in that list were needed to identify the variable in many of the XV calls. In Version 6, the calls to SAS_XV routines use the variable number from the list of variables in the data set and the specific list number is of little importance.

In Version 5, XVCOUNT returned the number of variables in the list. The equivalent routine in Version 6 is SAS_XOINFO with the code XO_NVARS, which returns the number of variables in the data set. This is the same as calling XVCOUNT in Version 5 for list 0.

Version 5 method

```
N = XVCOUNT(FILEID,0);
```

Version 6 method

```
CALL SAS_XOINFO(FILEID,XO_NVARS,
                ADDR(N));
```

In Version 5, you called XVDFLST to create a default list after you completed all the calls to XVLIST to create the lists specified by the user. The list created by XVDFLST omitted any variables specified on lists processed by XVLIST, so the placement of the call to XVDFLST was significant.

In Version 6, you now specify in the call to SAS_XVDFLST which lists to omit from the default list. The calling order for SAS_XVDFLST is no longer significant. You identify lists to omit by specifying pointers to the lists as the last arguments in the call.

Note: If you are writing your procedure to run under the VMS operating system, you must use the SAS_XVDFLST*n* routine, where *n* corresponds to the number of arguments.

Version 5 method

```
CALL XVLIST(FILEID,1,LISTPTR(1),
            NLIST(1));
CALL XVLIST(FILEID,4,LISTPTR(4),
            NLIST(4));
CALL XVLIST(FILEID,5,LISTPTR(5),
            NLIST(5));
CALL XVDFLST(FILEID,1,3);
```

Version 6 method

```
LISTPTR(1)=SFLP(proc.head,1);
LISTPTR(4)=SFLP(proc.head,4);
LISTPTR(5)=SFLP(proc.head,5);
CALL SAS_XVDFLST(FILEID,3,LISTPTR(1),
            2,LISTPTR(4),LISTPTR(5));
```

The Version 5 XVFIND routine looked up a variable in a specific list. In Version 6, SAS_XVFIND looks up the variable in the data set to see if it exists. If you need to verify that the variable exists in a specific list, use the variable number returned by SAS_XVFIND and compare that number to the numbers in the STLIST element for the list. This technique is illustrated in the description of SAS_XVFIND in Chapter 22.

The following set of examples shows how to look up variables in Version 5 and Version 6:

Version 5 method

```
VARNUM = XVFIND(FILEID,VARNAME,0);
IF VARNUM=0
   THEN CALL XLOG2('Variable # not
                found.',VARNAME);
```

Version 6 method

```
IF SAS_XVFIND(FILEID,ADDR(VARNAME),
         ADDR(VARNUM)) = 0
  THEN CALL SAS_XPSLOG2('Variable
         %s not found in data set.',
         VARNAME);
```

Scatter Reading and Gather Writing

The scatter-reading and gather-writing processes in Version 5 and Version 6 are very similar.

Changes to Scatter Reading

Changes to scatter reading include the following:

- □ You now end the scatter-read process with a call to SAS_XVGETE, a new routine.
- □ Instead of using different routines for offset versus absolute moves (as in the Version 5 XVGETD or XVGETDO and XVGET or XVGET2 routines), you use the SAS_XVGETD routine in Version 6 to define both types of moves and the SAS_XVGET routine to perform both types of moves. The codes for the types of move (formatted or otherwise) have changed.

The following set of examples illustrates the new scatter-reading techniques:

Version 5 method

```
CALL XVGETI(FILEID,3,XVGETPTR);
DO I=1 TO 3;
   CALL XVGETD(FILEID,6,VARNUM(I),
               ADDR(X(I)),8,'F');
   END;
...
CALL XVGET(XVGETPTR);
```

Version 6 method

```
CALL SAS_XVGETI(FILEID,3,XVGETPTR);
DO I=1 TO 3;
   CALL SAS_XVGETD(FILEID,VARNUM(I),
                   0,ADDR(X(I)),8,
                   XV_NOFMT);
   END;
CALL SAS_XVGETE(XVGETPTR);
...
CALL SAS_XVGET(XVGETPTR,NULO);
```

Changes to Gather Writing

The changes to gather writing include the following:

- □ Instead of using different routines for offset versus absolute moves (as in the Version 5 XVPUT or XVPUT2 routines), you use the SAS_XVPUT routine in Version 6 to perform both types of moves.
- □ The way in which you use the XVPUTD routine has changed.
- □ The call to SAS_XVPUT does not actually write an observation. You must call SAS_XOADD after SAS_XVPUT to write the observation.

These are explained in more detail in the following paragraphs.

In Version 5, you called the XVPUTD routine to define the location of variables to be output. If you were doing an absolute move, you provided a pointer in the call to XVPUTD and called XVPUT to move the observation. For an offset move, you set the pointer to NULL and provided the NPOS value in the NAMESTR. You then called XVPUT2 to move the observation. Both XVPUT and XVPUT2 actually wrote the observation to the output file.

In Version 6, with the SAS_XVPUTD routine, both the offset value and absolute pointer value are passed as arguments, and the NPOS value is not used for output data set definition. The absolute pointer is non-NULL for absolute moves, and NULL otherwise.

If all moves are absolute moves, the second argument to SAS_XVPUT, the base pointer for offset moves, is NULL. For offset moves, the second argument provides the base pointer for the offsets.

Because SAS_XVPUT no longer writes the observation, you can perform multiple sets of offset moves using different base pointers before writing an observation. The observation is not written to the output data set until you call the SAS_XOADD routine.

▶ *Caution* .... ***If you perform multiple sets of offset moves, be sure that you define a different control pointer with SAS_XVPUTI for each set of offset moves.***

If you attempt to use a single pointer, you will overwrite the previously gathered data. ▲

The following set of examples illustrates the new gather-writing techniques:

Version 5 method

```
CALL XVPUTI(FILEID,3,XVPUTPTR);
CALL XVPUTD(XVPUTPTR,ADDR(NAMESTR(1)),
            ADDR(CODE),8);

   /* NAMESTR(2) has NPOS */
   /* set to OFFSET       */
CALL XVPUTD(XVPUTPTR,ADDR(NAMESTR(2)),
            $NULL(),8);
CALL XVPUTD(XVPUTPTR,ADDR(NAMESTR(3)),
            ADDR(AGE),8);
CALL XVPUTE(XVPUTPTR);
....
CALL XVPUT2(XVPUTPTR,BASEPTR);
....
CALL XVPUTT(XVPUTPTR);
```

Version 6 method

```
NUL0 = SAS_NUL();
CALL XVPUTI(FILEID,3,XVPUTPTR);
CALL XVPUTD(XVPUTPTR,ADDR(NAMESTR(1)),
            0,ADDR(CODE),8);
CALL XVPUTD(XVPUTPTR,ADDR(NAMESTR(2)),
            OFFSET,ADDR(NAME),8);
CALL XVPUTD(XVPUTPTR,ADDR(NAMESTR(3)),
            0,ADDR(AGE),8);
CALL XVPUTE(XVPUTPTR);
....
CALL XVPUT(XVPUTPTR,BASEPTR);
CALL XOADD(FILEID,NUL0);
....
CALL XVPUTT(XVPUTPTR);
```

Chapter 18 Converting from Release 82.4 to Version 6

Introduction

A Release 82.4 procedure is completely different from a Version 6 procedure. To convert a Release 82.4 procedure to a Version 6 procedure, you must completely rewrite the procedure. This chapter contrasts the general concepts of procedure writing in the two releases and then lists each routine available in Release 82.4. The discussion of these routines should help you find the appropriate routines to use in Version 6, but you need to be thoroughly familiar with Part 2, "Grammars," and Part 3, "Procedures," of this book before you attempt to rewrite a Release 82.4 procedure as a Version 6 procedure. The examples in this chapter are in PL/I unless the routine being discussed is used only for another language.

Summary of Release 82.4 Procedure Writing

A Release 82.4 procedure consists of two parts: a parsing module and a procedure module. The parsing module is an assembler program containing special assembler macros, such as SASPROC and SASLIST. These macros define the syntax of the procedure. These modules can also include custom assembler coding to handle the syntax of unusual statements. The parsing module parses the user's statements and stores information about the statements in the statement structures. When the parsing is completed, the parser loads the procedure module, deletes itself, and passes control to the procedure module.

The procedure module steps through the statement structures built by the parsing module. Once all the statement structures are evaluated, the procedure begins operating on input and output SAS data sets. The procedure switches from one data set to another using the concept of the *current data set*. The procedure also manages the buffers between the various SAS data sets.

Printing to the log is handled by the SASLOG routine; any printing to the print file is done with the native printing routines for the given language (PUT statements in PL/I, WRITE or PRINT statements in FORTRAN). Pagination is handled by the STITLE routine.

Summary of Differences in Version 6

In contrast, Version 6 procedures handle parsing by using grammar tables. The grammar tables are created by compiling a special grammar module that defines the syntax of the procedure. The procedure module calls the parser and passes it the grammar table for parsing. One similarity between Release 82.4 and Version 6 is that the parser stores the results of parsing the user's statements in the statement structures. After parsing is completed, the procedure module regains control and accesses the statement structures to obtain information about how the user called the procedure. The Version 6 procedure accesses SAS data sets with a fileid pointer, so there is no need for any data set to be the current one. The SAS Supervisor handles all buffer management, so you do not need to be concerned with it.

To print procedure output to the log, use the SAS_XPSLOG routines, which allow for flexible formatting. To print to the procedure output file, you use a variety of SAS_XP routines. If you use native printing routines, your output cannot be routed to the procedure output file and, therefore, it cannot be displayed in the SAS Display Manager System's OUTPUT window.

Conversion of Individual Routines

The remaining sections in this chapter briefly describe the routines available in Release 82.4 and indicate if there is a comparable routine in Version 6.

ABUF

The ABUF routine allocated a buffer for the current data set. In Version 6, this is unnecessary, since buffer management is handled by the SAS Supervisor.

ADSTBL

The ADSTBL routine defined the name of a new SAS data set to be opened with OPENL. This is unnecessary in Version 6; the XOPNSTR passed to the SAS_XOOPEN routine defines the name of the data set to be opened.

BEST

The BEST routine formatted a number using the BEST. format. In Version 6, you can use the SAS_XFXPN routine with a format code of 0. The 0 format code corresponds to the BEST. format. The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
DCL NCHAR CHAR(12),
    NCHARF FLOAT BIN(53)
       BASED(ADDR(NCHAR));
CALL BEST(NUMBER,12,NCHARF);
```

Version 6 method

```
RC=SAS_XFXPN(NUMBER,12,0,0,
             ADDR(NCHAR));
```

BUFSIZ

The BUFSIZ routine queried the buffer size of the current data set. This is unnecessary in Version 6.

BYCOMP

The BYCOMP routine compared the BY groups of two different data sets. This is accomplished by the SAS_XBYSYNC routine in Version 6. The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
CALL SETDSN(1);
CALL GETDS(DS1PTR);
CALL SETDSN(2);
CALL GETDS(DS2PTR);
CALL BYCOMP(DS1PTR,DS2PTR,I);
```

Version 6 method

```
RC=SAS_XBYSYNC(FILEID1,FILEID2,RC);
```

BYLINE

The BYLINE routine accessed the current BY line. This capability is no longer available.

BYPASS

The BYPASS routine advanced to the next BY group. In Version 6, you can use the SAS_XBYNEXT routine, which is now a function instead of a subroutine. The function format is most useful in a DO WHILE statement. The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
CALL BYPASS(I);
DO WHILE(I¬=0);
   BY-group processing
   CALL BYPASS(I);
   END;
```

Version 6 method

```
DO WHILE(SAS_XBYNEXT(FILEID) ¬= 0);
   BY-group processing
   END;
```

CLOSE

The CLOSE routine closed the current data set. In Version 6, the corresponding routine is SAS_XOCLOSE. The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
CALL CLOSE;
```

Version 6 method

```
RC=SAS_XOCLOSE(FILEID,0);
```

CLOSED

The CLOSED routine closed a utility file and deleted it. In Version 6, the SAS_XCCLOSE routine with the XC_DELETE flag accomplishes this. The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
CALL CLOSED;
```

Version 6 method

```
RC=SAS_XCCLOSE(FILEID,XC_DELETE);
```

CLSOUT

The CLSOUT routine closed an output data set after processing. In Version 6, you can use the SAS_XOCLOSE routine. The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
CALL CLSOUT;
```

Version 6 method

```
RC=SAS_XOCLOSE(FILEID);
```

COMADD

The COMADD routine returned the address of the prologue code. This concept does not exist in Version 6. If you have this call in your code, you must redesign your application using the instructions in Part 2 and Part 3 of this book.

DLTOUT

The DLTOUT routine deleted the modules used in processing the output data sets. In Version 6, output data sets are handled differently, so this routine is unnecessary.

DSINIT

The DSINIT routine prepared a utility file for reuse. In Version 6, you can prepare a utility file for reuse by closing it with SAS_XCCLOSE and then opening it again with SAS_XCOPEN.

EFFMT

The EFFMT routine formatted a number using the E format, similar to the way BEST used the BEST format. In Version 6, you can load the E format with SAS_XFNAME, then call SAS_XFXPN with the format code. The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
DCL  NCHAR CHAR(12),
     NCHARF FLOAT BIN(53)
        BASED(ADDR(NCHAR));
CALL EFFMT(VALUE,12,2,NCHAR);
```

Version 6 method

```
RC=SAS_XFNAME('E        ','F',
              SAS_NUL(),ADDR(FCODE));
RC=SAS_XFXPN(VALUE,12,2,FCODE,
             ADDR(NCHAR));
```

ENDNAM

The ENDNAM routine signaled the end of the definitions of output variables. In Version 6, you can use the SAS_XVPUTE routine. The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method	Version 6 method
calls to ONAME2 and ONAMES	*calls to SAS_XVPUTD*
`CALL ENDNAM;`	`RC=SAS_XVPUTE(XVPUTPTR);`

EXECx Routines

The EXEC, EXECI, EXECR, and EXECT routines executed compiled code. This mechanism is not supported in Version 6. If your application uses these routines, you must rewrite the application to remove them.

FBUF

The FBUF routine freed a buffer previously allocated with ABUF. Since buffer management is handled by the SAS Supervisor in Version 6, this routine is unnecessary.

FDSTBL

The FDSTBL routine freed a data set table allocated by ADSTBL. Since ADSTBL is no longer used, this routine is unnecessary.

FORMAT

The FORMAT routine formatted a variable using the format associated with the variable. In Version 6, you can use the SAS_XFXPN and SAS_XFXPC routines to format numeric and character variables, respectively, or you can use the SAS_XVGETD or SAS_XVGET1V routine, depending on your application.

Note: In Version 6, you are not limited to formatting a variable with the format associated with the variable in the name descriptor.

The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
CALL FORMAT(0,N,CDATA(1),LNG);
```

Version 6 method

```
RC=SAS_XVGET1V(FILEID,N,LNG,
               ADDR(CDATA));
```

FREEMEM

The FREEMEM routine freed memory allocated by GETMEM. In Version 6, you can use either the SAS_XMEMFRE or SAS_XMFREE routine, depending on how you allocated the memory (via SAS_XMEMGET or SAS_XMPOOL).

GETDS

The GETDS routine obtained the data set table address for the current data set. In Release 82.4, you could process only one data set at a time. The data set you were processing was the *current data set*. You called the SETDSN to establish the current data set. Version 6 uses file pointers to enable you to process any open file by simply specifying the fileid pointer. Since there is no current data set concept in Version 6, there is no need for this routine.

GETDSL

The GETDSL routine obtained the logical data set table address for a utility file. Since the concept of the current data set does not exist in Version 6, there is no need for this routine.

GETL

The GETL routine read a record from a utility file. You read a utility file in Version 6 with the SAS_XCREAD routine. Note that in Version 6, you must give the number of bytes you want to read. The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method	**Version 6 method**
`CALL GETL(AREA(1));`	`RC=SAS_XCREAD(FILEID,ADDR(AREA), NBYTES);`

GETMEM

The GETMEM routine dynamically allocated memory. In Version 6, the equivalent routine is SAS_XMEMGET. The GETMEM routine used an unusual calling sequence that indicated the number of bytes to allocate, the number of bytes actually allocated (returned), the pointer to the allocated memory (returned), and the amount to leave unallocated in the partition. The LEAVE argument is now irrelevant in Version 6. Also, in Version 6, you obtain either all the memory you request or none of it, so you cannot determine if a partial allocation is possible.

The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method	**Version 6 method**
`CALL GETMEM(100,ABYTES,MEMPTR,0);`	`MEMPTR = SAS_XMEMGET(100);`

GETOBS

The GETOBS routine read observations without concern for BY processing. It returned the pointer to the observation buffer, or NULL if it reached the end of file. In Version 6, the equivalent routine is SAS_XOGETN followed by a call to SAS_XOINFO. You must call SAS_XOINFO to obtain the observation buffer address because SAS_XOGETN does not provide this.

The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method	**Version 6 method**

`CALL GETOBS(PTR);`	`RC=SAS_XOGETN(FILEID);`
	`RC=SAS_XOINFO(FILEID,XO_CUROBS,`
	`ADDR(PTR));`

GETSTR

The GETSTR routine allocated memory in FORTRAN procedures and returned an index into a BASE array. In Version 6, use the SAS_XM routines, whichever are appropriate, to allocate memory. Then, use the LOCPTR routine to obtain the BASE array index.

The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
CALL GETSTR(NEED,BASE,INDX,ISW,LEAVE)
```

Version 6 method

```
INTEGER*4 P,NEED
REAL*8 BASE(0:1)

P = SAS_XMEMEX(NEED)
INDX = LOCPTR(BASE,8,P)
```

HISTRY

The HISTRY routine transferred history records from an input to an output data set. Release 6.07 and earlier Version 6 releases do not have history records, so there is no equivalent routine.

IADDR

The IADDR routine permitted FORTRAN code to obtain the address of entities, returning the address in an INTEGER*4. This routine is also available in Version 6 with the same calling convention. Note that if you want to write portable FORTRAN code, do not use the IADDR routine with character variables or constants. Use the CADDR routine instead. Refer to Chapter 24, "Miscellaneous Routines and Values," for a detailed description of IADDR and CADDR.

INPUT

The INPUT routine advanced to the next observation in the input data set, honoring BY processing. In Version 6, use the SAS_XBYGET routine to advance to the next observation. The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
CALL BYPASS(I);
DO WHILE(I¬=0);
   CALL INPUT(IEND);
   DO WHILE(IEND¬=1);
      process observation
      CALL INPUT(IEND);
      END;
   ...
   END;
```

Version 6 method

```
DO WHILE(SAS_XBYNEXT(FILEID)¬=0);
   DO WHILE(SAS_XBYGET(FILEID)¬=1);
      process observation
      END;
   BY-group processing
   END;
```

IOPT

The IOPT routine indicated which bit options appeared in the user's PROC statement. All bit options had numbers, and these options could be accessed via IOPT. IOPT was also used to obtain some SAS system options, such as OVP (43) and CENTER (50). In Version 6, you locate options that can be specified in your procedure's statement in the statement structure. Use the SFOPT routine to access the option settings for your procedure. For SAS system options, use the SAS_XOPTBGT, SAS_XOPTCGT, and SAS_XOPTIGT routines.

LINSIZ

The LINSIZ routine indicated the current line size. In Version 6, you can call SAS_XPOPTGT with the XP_LINESIZE mnemonic. The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
CALL LINSIZ(LINESIZE);
```

Version 6 method

```
LINESIZE=SAS_XPOPTGT(XPLINESIZE,
                     ADDR(RC));
```

LISTDF

The LISTDF routine added extra lists of variables to the data set. In Version 6, this is not necessary. SAS_XVDFLST uses lists of variables to create a default list, but all other references to variable numbers relate to the variables in a data set, not to variables in a specific list.

LOCPTR

The LOCPTR routine permitted FORTRAN programs to obtain an index for a location in dynamically allocated memory, based off a supplied BASE array. This routine is still available in Version 6 and it functions as it did in Release 82.4. Refer to Chapter 24 for a detailed description of LOCPTR.

LODOUT

The LODOUT routine loaded the output processing routines, which could later be deleted with DLTOUT since they took a considerable amount of memory. This is now handled by the SAS Supervisor in Version 6.

LOGPRC

The LOGPRC routine redirected log output to the procedure output file. In Version 6, use the SAS_XPSPRN routine to print to the procedure output file and the SAS_XPSLOG routine to print to the log. These two routines use the same calling convention but print to different destinations.

LOGSYS

The LOGSYS routine returns control of log message printing to the log. In Version 6, use the SAS_XPSLOG routine.

MOVE

The MOVE routine moved data from one pointer location to another for a given length. The SAS_ZMOVEI routine performs this action in Version 6 and has the same calling sequence as MOVE. This routine is particularly useful in FORTRAN, which has no native way to deal with pointers.

The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method	**Version 6 method**
`CALL MOVE(FROM,TO,LENGTH);`	`CALL SAS_ZMOVEI(FROM,TO,LENGTH);`

MEMERR

The MEMERR routine printed a message indicating that memory could not be allocated. In Version 6, call the SAS_XPSLOG routine with a message. You can also use SAS_XMPOOLC with the XM_EXIT flag or SAS_XMEMEX to allocate the memory. These routines exit and print an error message about insufficient memory when the memory is unavailable.

NAMES and NAMEV

The NAMES routine obtained a variable's name, type, and length. The NAMEV routine obtained the entire NAMESTR (variable descriptor structure). In Version 6, SAS_XVNAME returns the NAMESTR information about a variable. Version 6 has no equivalent for NAMES; simply call SAS_XVNAME to obtain the NAMESTR and extract what you need.

The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
   /* obtain NAMESTR information */
CALL NAMEV(0,N,NTYPE);

   /* obtain specific information */
CALL NAMES(0,N,VNAME,ITYPE,LENGTH);
```

Version 6 method

```
   /* obtain NAMESTR information */
RC=SAS_XVNAME(FILEID,N,
              ADDR(NAMESTRPTR));

   /* extract specific information */
RC=SAS_XVNAME(FILEID,N,
              ADDR(NAMESTRPTR));
VNAME=NAMESTRPTR->NNAME;
ITYPE=NAMESTRPTR->NTYPE;
LENGTH=NAMESTRPTR->NLNG;
```

NORMINV

The NORMINV routine returned the associated normal random variable for a given probability. This function is available in Version 6 as the SAS_ZPROBIT routine.

Note: The SAS_ZPROBIT routine can be used with all languages, not just PL/I as was the case for NORMINV.

NOTE

The NOTE routine obtained the observation number for the most recently read observation. In Version 6, you can use the SAS_XONOTE routine. The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
CALL NOTE(IOBS);
```

Version 6 method

```
RC=SAS_XONOTE(FILEID,ADDR(IOBS));
```

NOTEP and NOTEPN

The NOTEP and NOTEPN routines recorded information about the current position in a variable-length utility file that was later used by the POINTP routine to reposition in the file. In Version 6, all utility files are byte-oriented, so extensive positioning information is not needed. You can use SAS_XCNOTE to note the current location in any utility file.

Note: Calling SAS_XCNOTE gives you the location of the next record in the file. In Release 82.4, NOTEP gave you the position of the most recently read record, and NOTEPN gave you information on the next record. The Version 6

SAS_XCNOTE routine corresponds to NOTEPN. To mimic the NOTEP action, call SAS_XCNOTE before you read the file with SAS_XCREAD.

The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method	Version 6 method
`CALL NOTEP(DATA);`	`RC=SAS_XCNOTE(FILEID,ADDR(DATA));`

NOVAR

The NOVAR routine indicated the number of variables in a list for the current data set. You can get the number of variables in any list in Version 6 by calling the SFN routine for the statement structure element for the list.

OBSERR

The OBSERR routine printed a message when the input data set had no observations. Version 6 prints a message automatically when you call the SAS_XOINFO routine with the XO_ANY code.

OBSPTR

The OBSPTR routine returned the pointer to the current observation buffer. In Version 6, call the SAS_XOINFO routine with the XO_CUROBS code. The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
CALL OBSPTR(DSPTR);
```

Version 6 method

```
RC=SAS_XOINFO(FILEID,XO_CUROBS,
              ADDR(DSPTR));
```

ONAMES

The ONAMES routine defined variables to the output data set. In Version 6, you can use the SAS_XVPUTD routine. The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
   /* absolute move */
CALL ONAME2(NTYPE,VECTOR(1),VAR);

   /* offset move */
CALL ONAMES(NTYPE,VECTOR(1));
```

Version 6 method

```
   /* absolute move */
RC=SAS_XVPUTD(XVPUTPTR,ADDR(NAMESTR),
              0,ADDR(VAR),
              NAMESTR.NLNG);

   /* offset move */
RC=SAS_XVPUTD(XVPUTPTR,ADDR(NAMESTR),
              NAMESTR.NPOS,
              SAS_NUL(),NAMESTR.NLNG);
```

OPENL

The OPENL routine opened a utility file. In Version 6, you can use the SAS_XCOPEN routine. The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method	Version 6 method
`CALL OPENL(DSNAME);`	`RC = SAS_XCOPEN(ADDR(FILEID), ADDR(XOPNSTR));`

OPNOUT

The OPNOUT routine opened the output data set for processing. In Version 6, the output data set is usually opened during parsing, as a result of the @DS semantic action. If you need to open the data set within the procedure, use the SAS_XOOPEN routine.

PAGSIZ

The PAGSIZ routine determined the current page size. In Version 6, use the SAS_XPOPTGT routine with the XP_PAGESIZE code. The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method	Version 6 method
`CALL PAGSIZ(PAGESIZE);`	`PAGESIZE = SAS_XPOPTGT(XPPAGESIZE, ADDR(RC));`

PARM

The PARM routine accessed information from the user's statement structures. In Version 6, use the appropriate SF routine.

PGF, PGT2, and PGX2 (Probability Functions)

The PGF, PGT2, and PGX2 routines computed probabilities. These routines correspond to the Version 6 routines as follows:

- SAS_ZPRBF replaces PGF.
- SAS_ZPGT2 replaces PGT2.
- SAS_ZPGX2 replaces PGX2.

POINT

The POINT routine positioned to an observation in a SAS data set or to a record in a fixed-length utility file. The position was obtained by a prior NOTE call. In Version 6, you can call the SAS_XOPOINT routine for SAS data sets and the SAS_XCPOINT routine for utility files. SAS_XOPOINT uses a record ID to indicate position. The record ID is obtained by a prior SAS_XONOTE call. SAS_XCPOINT expects a position value that is a long integer returned by a prior SAS_XCNOTE call.

The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method	**Version 6 method**
`CALL POINT(INT_POS);`	`RC=SAS_XOPOINT(FILEID,ADDR(RID));`

POINTP

The POINTP routine positioned to a location in a variable-length utility file. The position was obtained by a prior NOTEP or NOTEPN call. In Version 6, call the SAS_XCPOINT routine using the long-integer value returned by SAS_XCNOTE. The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
CALL POINTP(DATA);
```

Version 6 method

```
RC=SAS_XCPOINT(FILEID,
               POSITION_VALUE);
```

PUNCH

The PUNCH routine checked whether the CARDS system-parameter value had been exceeded. This routine is not supported in Version 6.

PUTL

The PUTL routine wrote a record to a utility file. The corresponding Version 6 routine is SAS_XCWRITE. In addition to writing records, the PUTL routine could also update a record in the observation buffer and write that record to the file. You can still update a record in a file in Version 6, but it performs the update by moving the record to your user area, where it is changed and written out; that is, the update is by move mode, not locate mode.

The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
CALL PUTL(AREA(1),NCHAR);
```

Version 6 method

```
RC=SAS_XCWRITE(FILEID,ADDR(AREA),
               NCHAR);
```

PUTOUT

The PUTOUT routine performed a gather write on the data specified in prior ONAMES calls and wrote the observation to the output data set. The PUTOUT routine used the same vector specified as an argument to the various ONAMES calls. In Version 6, you can use the SAS_XVPUT routine to perform the gather write, using the pointer first provided in the SAS_XVPUTI call. To actually write the observation, use the SAS_XOADD routine.

The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
CALL LODOUT;
CALL OPNOUT;
ensure VECTOR is at least 6*nvar+42
   bytes in size
call ONAMES using VECTOR

   /* all absolute moves */
CALL PUTOUT(VECTOR(1));
   /* at least one offset move */
CALL PUTOUT(VECTOR(1),BASEPTR);
```

Version 6 method

```
RC=SAS_XVPUTI(FILEID,nvar,XVPUTPTR);
call SAS_XVPUTD using XVPUTPTR

   /* all absolute moves */
RC=SAS_XVPUT(XVPUTPTR,SAS_NUL());
   /* at least one offset move */
RC=SAS_XVPUT(XVPUTPTR,BASEPTR);
   /* write record */
RC=SAS_XOADD(FILEID,XVPUTPTR);
```

RLABEL

The RLABEL routine obtained the libref, member name, label, type, and degrees of freedom of the data set. In Version 6, all of these values but the degrees of freedom can be obtained by calling SAS_XOINFO with the appropriate codes. The degrees-of-freedom global value is not supported in Version 6.

The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
CALL RLABEL(DSNAME(1),DSLABEL(1));
```

Version 6 method

```
   /* to access the libref */
RC=SAS_XOINFO(FILEID,XO_LIB,
                    ADDR(DSNAME(1)));
   /* to access the member name */
RC=SAS_XOINFO(FILEID,XO_MEM,
                    ADDR(DSNAME(2)));
   /* to access the type */
RC=SAS_XOINFO(FILEID,XO_TYPE,
                    ADDR(DSLABEL(2)));
   /* to access the label */
RC=SAS_XOINFO(FILEID,XO_LABEL,
                    ADDR(DSLABEL(3)));
```

RWND

The RWND routine rewound to the beginning of the current BY group. In Version 6, you specify 'B' (indicating BY-group rewinding) as the second argument to the SAS_XBYRWND routine to rewind a BY group. You use the same routine, with an argument of 'F', to rewind to the beginning of the data set.

The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method	Version 6 method
`CALL RWND;`	`RC=SAS_XBYRWND(FILEID,'B');`

RWNDBY

The RWNDBY routine positioned to the first observation of the current data set, ignoring BY groups. In Version 6, you specify 'F' (indicating the first observation of the data set) as the second argument to the SAS_XBYRWND routine to rewind a data set to the beginning. In fact, SAS_XBYRWND rewinds the data set to the first observation to be processed in the data set; therefore, if the user has specified the FIRSTOBS= option, the data set is rewound to this observation.

The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method	Version 6 method
`CALL RWNDBY;`	`RC=SAS_XBYRWND(FILEID,'F');`

SASFT, SASFVS, SASPLF(N), and SASPLO(N)

These interface routines were used in Release 82.4 to establish the proper language environment. In Version 6, the first executable statement must be a call to UWPRCF for FORTRAN, UWPRCP for PL/I, or UWPRCC for C and IBM 370 assembler. The SASPLFN and SASPLON routines were used to avoid opening the print file when there was no output. These routines are no longer necessary with the Version 6 SAS_XP printing routines.

SASLOG

The SASLOG routine wrote messages to the log. You were responsible for formatting your own message to be printed. Two extra features available in SASLOG were

- the ability to remove the ¬ character from the text if a ¬ appears as the last character of the text. This removed additional blanks, by changing the additional blanks to ¬ characters.
- the ability to indicate continuation lines for messages that spanned multiple lines. When messages spanned multiple lines, you made multiple calls to SASLOG. For all but the first line, the NOTE or ERROR was followed by a dash (—) instead of a colon (:). This caused the message to be indented without repeating the NOTE or ERROR string.

In Version 6, you use the SAS_XPSLOG*x* routines to print messages to the log. The message is automatically spanned across multiple lines and indented if necessary. Use the %1z substitution character instead of ERROR:, the %2z substitution character instead of WARNING:, and %3z instead of NOTE:. These substitution characters produce the appropriate prefix to the message and also ensure that continued lines are indented correctly.

The message length that was required in Release 82.4 is not used in calls to the SAS_XPSLOG routines. The conversion of '' to a null character is not supported in Version 6, but this feature is not really needed, since other directives for the SAS_XPSLOG routines such as %b can deal with trailing blanks. Note also that the Release 82.4 SASLOG routine required you to put a carriage-control character (usually blank) in the first byte of the log message. This is not necessary with Version 6.

The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
   /* simple note */
CALL SASLOG(' NOTE: A MESSAGE',16);

   /* note with substitution value */
PUT STRING(ABC) EDIT(VALUE)(F(4));
ABC=TRANSLATE(ABC,'¬',' ');
CALL SASLOG(' NOTE: VALUE IS '||ABC,20);

   /* note with substitution */
   /* values split over two lines */
CALL SASLOG(' NOTE: LINE 1.',14);
CALL SASLOG(' NOTE- LINE 2.',14);
```

Version 6 method

```
   /* simple note */
CALL SAS_XPSLOG1('%3z A MESSAGE');

   /* note with substitution value */
CALL SAS_XPSLOG2('%3zVALUE IS %d',
                 VALUE);

   /* note with substitution */
   /* values split over two lines */
CALL SAS_XPSLOG3('%3z%s%n%s',
                 'MESSAGE LINE 1.',
                 'MESSAGE LINE 2.');
```

SETDS, SETDSL, and SETDSN

The SETDS, SETDSL, and SETDSN routines changed the current data set. In Version 6, all SAS data set and utility file processing is handled by using fileid parameters and the concept of the current data set is no longer relevant.

SETERR

The SETERR routine set the procedure condition code to indicate an error condition. In Version 6, you can call SAS_XEXIT with a return code other than XEXITNORMAL. The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
CALL SETERR;
```

Version 6 method

```
CALL SAS_XEXIT(XEXITERROR,0);
```

STITLE

The STITLE routine flushed the printing of title lines, provided information concerning printing, and in some cases set printing options. The LINES value returned by STITLE was the number of lines left on the page that were available to the procedure. The ITYPE value contained the settings of SAS system options. For example, if you called STITLE as follows, it returned the settings of some SAS system options in the ITYPE argument:

```
CALL STITLE(ITYPE,LINES);
```

You could also use the LINES argument to set options, such as the flag to suppress BY lines.

ITYPE contained a value that was a sum of any of the following values:

Value	Meaning
2	no page numbers
4	no date
8	do not print data set name
16	do not print BY lines
32	no titles
64	no page number incrementing
128	set page number to LINES value
512	set LINES to current page number

In Version 6, you can call the SAS_XPOPTGT routine with the XPLINESLEFT mnemonic to determine how many lines are left on a page. The SAS system options previously stored as a combined value in ITYPE can be retrieved individually in Version 6. Titles and footnotes are handled automatically for you, so you do not need to know if the user specified NOTITLES. You retrieve the settings of the NONUMBER and NODATE system options with SAS_XOPTBGT in Version 6. If you need to suppress BY lines, call SAS_XPOPTGT with the XPNOBYLINE mnemonic. All other options do not exist in Version 6.

UBUF

The UBUF routine allowed a specific area of memory to be used for a data set buffer. Since all buffer management is handled by the SAS Supervisor in Version 6, this routine is not needed.

UNIT

The UNIT routine obtained the FORTRAN-based unit numbers for the various SAS System files. The procedure used the unit number to build a DDname of the form FT*nn*F001 where *nn* was the IUSE value, as illustrated here:

```
CALL UNIT(2,IUSE);
OPEN FILE(SYSPRINT) TITLE('FT'||SUBSTR(CHAR(IUSE+100),13)||
                          'F001');
```

These steps are unnecessary in Version 6. In Version 6, the punch and plot files are no longer used by procedures. You can use the SAS_XP routines to print output directly to the SAS log or procedure output file, which are handled by the SAS Supervisor, so you do not need to open the file.

To obtain the PARMCARDS file, call SAS_XOPTCGT with the PARMCARDS option. The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
CALL UNIT(5,IUSE);
PARMDD='FT'||SUBSTR(CHAR(IUSE+100),
        13)||'F001';
```

Version 6 method

```
P = SAS_XOPTCGT("PARMCARDS",
                ADDR(RC));
CALL SAS_ZMOVEI(P,ADDR(PARMDD),8);
```

UNUSED

In Release 82.4, many SAS procedures dynamically allocated the largest contiguous block of unused memory for their own use. The UNUSED routine told the SAS Supervisor the amount of memory allocated but not used by a procedure. When the procedure reported the amount of memory not used, then the SAS Supervisor could determine the amount actually used by the procedure.

In Version 6, the SAS System no longer allocates all possible memory. Your procedure needs to estimate the needed memory and allocate memory based on those estimates.

VAR

The VAR routine transferred a single variable into memory, with a limitation of 8 bytes of memory. Therefore, numeric values could be successfully transferred, but character variables over 8 bytes were truncated.

In Version 6, the equivalent routines are the SAS_XVGET1F routine for numeric data and SAS_XVGET1C for character data. With SAS_XVGET1C, character variables are not truncated (unless you want them to be).

Note: Use these routines only in special situations because they are much less efficient than the SAS_XVGETD scatter-read routines. The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
DCL VALUE FLOAT BIN(53),
    VALUEC CHAR(8)
       BASED(ADDR(VALUE));

VALUE = VAR(0,N);
   /* VALUEC has character value */
VALUEC = VAR(0,N+1);
```

Version 6 method

```
DCL VALUE FLOAT BIN(53),
    VALUEC CHAR(8);

RC=SAS_XVGET1F(FILEID,N,
                    ADDR(VALUE));
RC=SAS_XVGET1C(FILEID,N,8,
                  ADDR(VALUEC));
```

VARERR

The VARERR routine printed a message when there were no variables in the data set. In Version 6, you can use the SAS_XOINFO routine with the XO_ANY code. It prints an appropriate message and returns a −1 return code. The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
CALL VARERR;
```

Version 6 method

```
RC=SAS_XOINFO(FILEID,XO_ANY,
                 ADDR(RC));
```

VARX

The VARX routine transferred an entire list of variables into memory. The limitations were that all variables were limited to 8 bytes in length, and that all the variables had to be written contiguously in memory.

In Version 6, you can use the SAS_XVGET routines to scatter read variables. Call SAS_XVGETI to indicate the number of variables; then call SAS_XVGETD for each variable; and, finally, call SAS_XVGETE to terminate the process. Once you have finished defining the scatter-read process, call SAS_XVGET to retrieve the values. Any mix of numeric and character variables and lengths is permitted. Variable values do not need to be stored contiguously.

The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
DCL X(5) FLOAT BIN(53);

CALL VARX(0,X(1));
```

Version 6 method

```
DCL X(5) FLOAT BIN(53);

RC=SAS_XVGETI(FILEID,5,XVGETPTR);
DO I=1 TO 5;
   RC=SAS_XVGETD(XVGETPTR,I,0,
                 ADDR(X(I)),8,
                 XV_NOFMT);
   END;
RC=SAS_XVGETE;

DO WHILE(SAS_XBYNEXT(FILEID)¬=0);
   DO WHILE(SAS_XBYGET(FILEID)¬=1);
      RC=SAS_XVGET(XVGETPTR);
      process observation
      END;
   END;
```

VARY

The VARY routine was the equivalent of the VARX routine, except that it permitted you to read an entire character variable, up to the limit of 200 characters. In Version 6, the SAS_XVGET1C and SAS_XVGETD routines permit this. See the discussion in VARX for more details.

WLABEL

The WLABEL routine wrote out the data set label, type, and degrees-of-freedom values. In Version 6, there is no global degrees-of-freedom value. To write the label and type, call the SAS_XOINFO routine with the XO_SLABEL and XO_STYPE codes. You can also specify the label and type in the XOPNSTR if you open a data set for output from within your procedure. In most cases, however, the data set has already been opened during parsing.

The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method

```
CALL WLABEL(DSLABEL(1));
```

Version 6 method

```
RC=SAS_XOINFO(FILEID,XO_SLABEL,
              ADDR(DSLABEL(3)));
RC=SAS_XOINFO(FILEID,XO_STYPE,
              ADDR(DSLABEL(2)));
```

XCTL

The XCTL routine transferred control to the next module. Control did not return to the calling module.

In Version 6, the closest equivalent is SAS_XZOSCMD. This routine allows you to invoke a command, similar to how the X command works with the SAS

System interactively. On MVS, if you use the XEQ: qualifier with the command string, you can invoke it in batch. Otherwise, SAS_XZOSCMD can only be used interactively. On CMS and VMS, SAS_XZOSCMD works either in batch or interactive mode. When the command has finished executing, SAS_XZOSCMD returns control to your procedure.

The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method	**Version 6 method**
`CALL XCTL('ABCD    ');`	`RC=SAS_XZOSCMD('ABCD    ', ADDR(RC));`

Note: Keep in mind that when you use the SAS_XZOSCMD routine, your program is not portable.

ZERO

The ZERO routine set to 0 section of memory for a given number of bytes. In Version 6, the SAS_ZZEROI routine does this. The following example compares the Release 82.4 and Version 6 processes:

Release 82.4 method	**Version 6 method**
`CALL ZERO(PTR,L);`	`CALL SAS_ZZEROI(PTR,L);`

Informats, Formats, Functions, and CALL Routines (IFFCs)

Unlike procedures, most IFFCs written for releases of the SAS System under MVS and CMS prior to Version 6 can still function under Version 6 without change. Keep in mind, however, that IFFCs written for earlier releases of the SAS System are not portable to any operating systems other than MVS and CMS.*

If you choose to rewrite an IFFC in the Version 6 style, the conversion of these routines is less complicated than converting procedures because these routines are much more limited in scope. In most cases prior to Version 6, formats and informats were written only in assembler language because the register calling convention was somewhat unorthodox:

R1 pointed to the data to format or informat

R2 contained the length of data−1 (for the EX instruction)

R3 contained the resulting length

R4 contained the number of decimals

* In Version 5, you could also write informats, formats, and functions under VMS, but those modules are not portable to any other operating system. You also cannot continue to write informats, formats, or functions using the Version 5 process. However, you can use the V5TOV6 procedure in base SAS software to convert those modules, so they can continue to function with Version 6.

In Version 6, you can use any language supported for SAS/TOOLKIT software to write IFFCs.

Functions used exactly one argument per SAS language function argument. Character arguments were supplied via a *rich pointer*, which contains a length in the first byte of the pointer, with the remaining 24 bits indicating the actual address. (This was meant for use under MVS and CMS operating systems prior to the XA family.) The function writer had to dereference this pointer to obtain the length and real pointer. In Version 6, character arguments are handled by passing three values for each character argument.

Prior to Version 6, IFFCs did not support calls back to the SAS Supervisor. In Version 6, if you initialize the SAS environment by calling UWPRCC, UWPRCF, or UWPRCP, you can use any of the SAS_X or SAS_Z routines.

REFERENCE

Part 6
Specifications

Chapter 19 The USERPROC Procedure

Introduction

The USERPROC procedure serves two purposes:

- It translates the grammar coding for a procedure into a function in the C, FORTRAN, PL/I, or IBM 370 assembler language that your procedure can use when calling the parser. The parser uses the syntax information in your grammar function to parse the user's input statements.
- It creates the program constants object file, which is necessary when linking all your object files into an executable module. Only user-written procedures need a grammar function, but all user-written SAS modules need a program constants object. The program constants object is unique for each executable module, so it must be re-created for each new module.

Specifications

The PROC USERPROC statement is the only statement associated with the USERPROC procedure:

PROC USERPROC <*grammar-args*> <*program-constants-args*>;

You use the USERPROC procedure with at least one set of options. You can create both a grammar function and a program constants object file in a single step of PROC USERPROC. The *grammar-args* specify the necessary files and additional options when you are running PROC USERPROC for a grammar. Use the *program-constants-arg* when you are running PROC USERPROC for a procedure, informat, format, function, or CALL routine. These sets of arguments are described in the following sections.

Grammar Arguments

When you use the USERPROC procedure to create a grammar function, you must specify these arguments:

- FUNCFILE=
- GRAMFUNC=
- GRAMMAR=.

There are several optional arguments you can use as well. All of the arguments for creating a grammar function are described below.

FUNCFILE=*fileref* | *'external-file'*
: gives the location of the resulting grammar function file written in the requested language. This argument is required if you are creating a grammar function.

GRAMLANG=C | FORTRAN | ASM370 | PLI
: gives the language for the resulting grammar function. If you omit this option, the default language is C. The supported compilers are as follows:

GRAMLANG=C
: generates code to be compiled using the SAS/C Compiler under CMS or MVS or the native C compiler under VMS.

GRAMLANG=FORTRAN
: generates code to be compiled by the VS FORTRAN compiler under CMS or MVS or the native FORTRAN compiler under VMS.

GRAMLANG=ASM370
: generates code to be assembled by the IBM 370 assembler under CMS or MVS.

GRAMLANG=PLI
: generates code to be compiled by the PL/I V2 compiler under CMS or MVS or the native PL/I compiler under VMS.

GRAMFUNC=*grammar-name*
: names the grammar function generated by PROC USERPROC. This argument is required if you are creating a grammar function. It is simpler, although not required, to use the same name for the value of the GRAMFUNC= argument as you use for the member or filename of the FUNCFILE= argument. For example, on MVS, GRAMFUNC can be ABCG, and the FUNCFILE= can be 'USERID.MYPGMS.SRC(ABCG)'.
  The value specified for the GRAMFUNC= argument is the name you use in the call to SAS_XSPARSE in the main procedure's code.

GRAMMAR=*fileref* | *'external-file'*
: gives the location of the input file containing the grammar source statements. This argument is required if you are creating a grammar function. See Chapter 3, "Writing a Grammar," for instructions on writing a grammar and Chapter 21, "Grammar Language," for reference information on the syntax of grammar files.

INCLLIB=*fileref* | *'aggregate-storage-location'*

gives the location of a library that contains grammar files that are named in a %INCLUDE statement in the GRAMMAR= file. Note that you must use only the name of the aggregate storage location (the partitioned data set, MACLIB, or directory name). Do not specify a file or member name.

In most cases, this option is needed because you must have a %INCLUDE statement for the STUBGRM grammar file. This file is supplied with SAS/TOOLKIT software and contains definitions for standard productions and other grammar items needed by your grammar. In addition, you may have other %INCLUDE statements to modularize your grammar. If you need to include grammar files from more than one aggregate storage location, use a FILENAME statement to associate a fileref with multiple aggregate storage locations.* Then specify the fileref as the value of the INCLLIB= option.

NOPRINT

specifies that you do not want printed output from the grammar processor.

Program Constants Arguments

When you use the USERPROC procedure to create a program constants object file, you must specify these arguments:

- MODNAME=
- OBJECT=.

There are several optional arguments you can use as well. All of the arguments for creating a program constants object file are described below.

MODNAME=*name*

gives the name of the executable module you are creating. If you are creating a program constants object, this option must be specified. Under MVS and CMS this name must match the NAME value in the statements to the linkage editor. Under VMS, this name must match the name of the .EXE file to be created by the linker.

The value of *name* can be any valid module name if you are using PROC USERPROC for a user-written procedure. If you are creating an informat, the name of this module must begin with UWI. User-written formats must begin with UWF, and user-written functions or CALL routines must begin with UWU. The next five characters should be the same as the first five characters of name of the IFFC. For example, if your IFFC is named ABCDEFGH, the MODNAME= value should be UW*x*ABCDE, where *x* is I, F, or U as described earlier. If your IFFC is named XYZ, the MODNAME= value should be UW*x*XYZ.

* You can also use operating system commands to set up a fileref that is associated with multiple aggregate storage locations. For example, on MVS you can assign a DDname to a concatenation of partitioned data sets.

MODTYPE=PROC
MODTYPE=INFORMAT
MODTYPE=FORMAT
MODTYPE=FUNCTION
: gives the type of executable module that you are creating. A different program constants object is generated for each different MODTYPE= value. If you omit this option, the default value for MODTYPE= is PROC.

OBJECT=*fileref* | *'external-file'*
: gives the name of the program constants object file that is created by the USERPROC procedure. If you are creating a program constants object, this option must be specified.

Details

For all arguments that permit an external filename except INCLLIB=, you specify the name of an external file. For the INCLLIB= option, you specify the name of an aggregate storage location, such as a PDS or a directory. In all cases, you can specify a fileref instead of an external file or aggregate storage location. When a fileref is used in an argument, you must have previously established the fileref with a FILENAME statement or operating system commands. For example, on MVS you can use a DDname statement in JCL or the TSO ALLOCATE command to create a DDname. You can then use that DDname for the fileref.

Procedures and IFFCs

When you use the USERPROC procedure to process an IFFC or a procedure, you must link the output of PROC USERPROC with the compiled program. Refer to the appropriate appendix for your operating system for more information on compiling and linking IFFCs and procedures.

Grammars

When you use the USERPROC procedure to process a grammar, you must compile the output of PROC USERPROC and link that object module to the compiled procedure. Refer to the appropriate appendix for your operating system for more information on compiling and linking procedures.

▶ *Caution* .... ***You should not modify anything in the function generated by PROC USERPROC except perhaps to add comments.***
Any change in the function can cause odd syntax errors for anyone using the procedure. If you need to change your grammar, you must make the changes to the grammar source code and then rerun PROC USERPROC to build a new function. ▲

Any grammar function generated by PROC USERPROC can be transported to any other operating system where SAS/TOOLKIT software supports that language. Therefore, when you port a procedure to another operating system, it is not necessary to run PROC USERPROC under that operating system to rebuild the grammar function. You can develop your source on one operating system, and port the entire source as needed. The generated source may look different depending on whether the original host operating system is EBCDIC- or ASCII-based, but either style works correctly under all operating systems.

Examples

Use the following statements to create a FORTRAN grammar function called ABCG from the grammar source file identified by the fileref ABC. The INCLUDE fileref is associated with the aggregate storage location of files containing grammar source statements your grammar can include.

```
filename abc 'grammar-source-file';
filename include 'your-include-file SAS/TOOLKIT-stub-grammar';
filename fsource 'FORTRAN-source-file';

proc userproc gramfunc=abcg language=fortran grammar=abc
              incllib=include funcfile=fsource;
```

The following example creates a program constants object file for a procedure that is invoked with the name NEWPROC. The name of the executable module in this example is also NEWPROC. The object is put in the file referenced by PGMCONST.

```
filename pgmconst 'object-file-name';

proc userproc object=pgmconst modname=newproc modtype=proc;
```

You can find more illustrations of how to use the USERPROC procedure in the appendix for your operating system.

Chapter 20 Data Structures

Introduction

This chapter describes the data structures used by SAS/TOOLKIT software to communicate between SAS programs and the procedures you write.

FATTRSTR

Format Attribute Structure

Declaration

```
struct FATTRSTR {
       short        fwmin;         /* minimum length                  */
       short        fwmax;         /* maximum length                  */
       short        fwdef;         /* default length                  */
       short        fjust;         /* justification: 1=left 0=right   */
/*----following added for data set usage------------------------------*/
       short        fdmin;         /* minimum no. of decimals         */
       short        fdmax;         /* maximum no. of decimals         */
       short        fddef;         /* default no. of decimals         */
       short        fmdef;         /* default modifier                */
       } ;
```

Usage

1. Allocate or declare a copy of the FATTRSTR.
2. Call SAS_XFNAME and pass the address of the FATTRSTR in the *ptr-to-fattrstr* parameter.

FATTRSTR *continued*

Description

The format attribute structure (FATTRSTR) is completed when you issue a call to SAS_XFNAME and pass the address of a FATTRSTR structure you have allocated or declared. The values stored in this structure are seldom needed by procedures so, in most cases, you simply pass a NULL pointer in the call to SAS_XFNAME.

The contents of the FATTRSTR structure are as follows:

`fwmin` indicates the minimum width permitted for the format loaded by SAS_XFNAME.

`fwmax` indicates the maximum width permitted for the format loaded by SAS_XFNAME.

`fwdef` indicates the default width of the format loaded by SAS_XFNAME.

`fwjust` indicates the justification of the format loaded by SAS_XFNAME. The value 0 indicates left-justified data; 1 indicates right-justified data.

Note: User-defined formats for numeric variables can cause problems when the format should be left justified. If a value is specified that has no corresponding format, the value is right justified in the space of the largest format width. For example, if the format is

1 = TRUE
0 = FALSE

then the string of values, 0 1 2, is stored as follows:

FALSE
TRUE
00002.

To avoid problems, always use the SAS_ZSTRJLS routine to ensure that all left-justified values (FWJUS=0) are truly left justified.

`fdmin` indicates the minimum number of decimal places permitted.

`fdmax` indicates the maximum number of decimal places permitted.

`fddef` indicates the default number of decimal places.

`fmdef` is a reserved field.

MEMLIST

SAS Data Library Member List

Declaration

```
struct  MEMLIST {
char8   mem;
char8   ext;
int     flags;
} ;
```

Usage

1. Do not allocate or declare a MEMLIST.
2. Call SAS_XDNAME with a NULL pointer for the *memlist* parameter.
3. Subsequent calls to SAS_XDNAME can reuse the value returned in the *memlist* parameter in the first call to SAS_XDNAME.

Description

The SAS data library member list structure (MEMLIST) is used by SAS_XDNAME to pass information about the members of a SAS data library. SAS_XDNAME returns the address of an array of MEMLISTs. Each MEMLIST contains information about one member of that SAS data library. The array includes a MEMLIST for each member of the data library that has the same type as the *member-type* parameter specified in the call to SAS_XDNAME. If you specify ALL for the *member-type*, the array includes a MEMLIST for every member of the library. If the library is sequential, you can access only one MEMLIST at a time.

The contents of the MEMLIST structure are as follows:

`mem` indicates the member name.

`ext` indicates the member type. Your procedure can access only the DATA or VIEW members of a SAS data library.

`flags` is a reserved field.

NAMESTR

Variable Name Structure

Declaration

```
struct NAMESTR {
 /*-----fields from the data set----------------------------*/
 long   ntype;       /* type of variable 1=num 2=char       */
 long   nlng;        /* length of variable                  */
 char  *nname;       /* name of variable                    */
 long   namelen;     /* length of variable name             */
 char  *nlabel;      /* ptr to label                        */
 long   nlablen;     /* length of label                     */
 char   nform[8];    /* format name                         */
 long   nfl;         /* format width                        */
 long   nfd;         /* format number of decimals           */
 long   nfj;         /* 0=left justification, 1=right just  */
 char   niform[8];   /* informat name                       */
 long   nifl;        /* informat width                      */
 long   nifd;        /* informat number of decimals         */
 long   npos;        /* position of value in observation    */
 long   nvar0;       /* number of variable on the file      */
 /*-----not from data set, derived fields via SAS_XFFILE---*/
 long   nflen;       /* format width                        */
 long   nfcode;      /* format code for variable            */
 long   nifcode;     /* informat code for variable          */
};
```

Usage

1. Allocate or declare a copy of the NAMESTR if you are creating an output variable or want to make any changes to the contents of a NAMESTR. If you only need to read the information in the NAMESTR, your procedure does not need to allocate or declare a copy of the NAMESTR. The NAMESTRs are allocated during parsing so your procedure can access those created by the parser.
2. Call SAS_XVNAME with the address of a pointer to a NAMESTR for the *ptr-to-namestr-ptr* parameter.
3. In most cases, you reuse the pointer to the NAMESTR you created each time you call SAS_XVNAME. If you want to store the NAMESTR information for all variables in the data set, you need to allocate an array of NAMESTRs.

Description

The variable name structure (NAMESTR) contains the name and attribute information for variables in a SAS data set. When you open a SAS data set, by specifying the @DS semantic action in the grammar or by calling SAS_XOOPEN, the supervisor creates a NAMESTR for each variable that can be accessed in the data set. In most cases, all variables in the SAS data set are available for processing, but variables can be excluded by using the DROP= and KEEP= data set options when the procedure is invoked. In addition to the information provided when the data set is opened, the variable name structures are updated by calls to SAS_XFFILE and SAS_XFNAME.

The SAS_XV routines use the data in the structures to read and create SAS data sets. To locate where the variable name structures are stored, call SAS_XVNAME for each variable you want to access. You can then use fields such as `nlng`, the length of the variable, in calls to SAS_XVGETD and SAS_XVPUTD.

▶ ***Caution*** .... ***The pointer returned by the call to SAS_XVNAME points to the actual NAMESTRs for the input data set. You must not alter the contents of these NAMESTRs.***
If you need to change a NAMESTR for output data set processing, you must create a new structure, copy values from the input NAMESTR, and change the values in the new structure. Refer to the description of SAS_XVNAMEI in Chapter 22, "SAS_X Routines," for more information on this process. ▲

If you want to reuse a NAMESTR that is created by your program for several invocations of SAS_XVPUTD, you can call SAS_XVNAMEI to initialize the structure to blanks and zeros where appropriate. Do not attempt to initialize a NAMESTR for one of the variables in the input data set.

The fields of the NAMESTR are described in detail below:

`ntype`
: is the code indicating the type of the variable. Codes are

  1 numeric

  2 character.

`nlng`
: is the length of the variable. For input data sets, this value is determined from the LENGTH statement specified for the variable or by the default length for the type of variable. For variables created by your program, this value can range from 1 to 200 for character variables and from 2 to 8 for numeric variables.

`nname`
: is a pointer to the name of the variable.

`namelen`
: is the length of the variable's name.

`nlabel`
: is a pointer to the label for the variable if one was assigned. Test the value of `nlablen` to determine if the variable has a label.

`nlablen`
: is the length of the variable's label. If the variable has a label, this length is greater than 0.

`nform`
: is the name of the format specified in a FORMAT or ATTRIB statement that was in effect when the procedure was invoked. This field is blank if neither of these statements was used. It is also blank if the format specified in one of these statements was *w.d*. You must test both this field and `nfl` to determine if a FORMAT or ATTRIB statement was specified for this variable.

`nfl`
: is the length of the format specified in a FORMAT or ATTRIB statement that was in effect when the procedure was invoked. If this field contains a value greater than 0 and the `nform` field is blank, the format specified was in the *w.d* form. If this field is 0 and `nform` is not blank, a format with no specific length (for example, DATE.) was used. You must test both this field and

NAMESTR *(nfl continued)*

`nform` to determine if a FORMAT or ATTRIB statement was specified for this variable.

`nfd`
: is the number of decimals in the format specified in a FORMAT or ATTRIB statement.

`nfj`
: is the code indicating how the value is justified in the field. Codes are

  0 left justification

  1 right justification.

`niform`
: is the name of the informat specified in an INFORMAT or ATTRIB statement that was in effect when the procedure was invoked. This field is blank if neither of these statements was used. It is also blank if the informat specified in one of these statements was *w.d*. You must test both this field and `nifl` to determine if an INFORMAT or ATTRIB statement was specified for this variable.

`nifl`
: is the length of the informat specified in an INFORMAT or ATTRIB statement that was in effect when the procedure was invoked. If this field contains a value greater than 0 and the `niform` field is blank, the informat specified was in the *w.d* form. If this field is 0 and `niform` is not blank, an informat with no specific length (for example, DATE.) was used. You must test both this field and `niform` to determine if an INFORMAT or ATTRIB statement was specified for this variable.

`nifd`
: is the number of decimals in the informat specified in an INFORMAT or ATTRIB statement.

`npos`
: is the number of bytes from the beginning of the observation buffer to this variable.

`nvar0`
: is the relative position of the variable in the data set.

`nflen`
: is the length of the format. This field is set by SAS_XFFILE. If a format is associated with this variable in the input data set, the length of the format is stored here. If no format is associated with the variable in the data set, SAS_XFFILE checks to see if a FORMAT or ATTRIB statement was used to provide a format. If so, this field is set to the length of the specified format. If not, the default length for the type of the variable is stored here.

`nfcode`
: is the access code for the format. This field is set by SAS_XFFILE. If a format is associated with this variable in the input data set, the code of the format is stored here. If no format is associated with the variable in the data set, SAS_XFFILE checks to see if a FORMAT or ATTRIB statement was used to provide a format. If so, this field contains the code of the specified format. If not, the code for the default format for the type of the variable is stored here.

Note: Access codes are assigned each time the procedure is executed; therefore, no static list of codes is available.

`nifcode` is the access code for the informat. This field is set by SAS_XFFILE. If an informat is associated with this variable in the input data set, the code of the informat is stored here. If no informat is associated with the variable in the data set, SAS_XFFILE checks to see if an INFORMAT or ATTRIB statement was used to provide an informat. If so, this field contains the code of the specified informat. If not, the code for the default informat for the type of the variable is stored here.

Note: Access codes are assigned each time the procedure is executed; therefore, no static list of codes is available.

PROC

Parsing Structure

Declaration

```
struct PROC {
   stmtptr      head;          /* proc statement              */
   stmtptr      last;          /* last statement              */
   stmtptr      current;       /* current statement           */
   short        error;         /* error flag                  */
   char8        name;          /* proc name                   */
   ptr          curfile;       /* current file                */
   ptr          lookfile;      /* current file to look up     */
                               /* variables                   */
   short        listnum;       /* current statement list number */
   char8        firstvarname;  /* first variable name         */
   char8        lastvarname;   /* last variable name          */
   double       dostart;       /* start for do lists          */
   double       doend;         /* end for do lists            */
   double       doby;          /* increment for do lists      */
   int          (*custom)();   /* pointer to custom semantic  */
                               /* action                      */
   short        lastn;         /* elements on list            */
   ptr          userptr;       /* communication for user      */
   ptr          procptr;       /* communication for proc & zsem */
   };
```

Usage

1. The parser allocates and uses the parsing structure as it parses the user's input; therefore, you do not need to allocate or declare a PROC structure.

2. Call SAS_XSPARSE with these parameters:

```
rc = SAS_XSPARSE(your-grammar-function, NULL, &proc);
```

PROC *continued*

3. The `proc.head` field contains the address of the first statement structure after the call to SAS_XSPARSE.

Description

The parsing structure (PROC) is used to store information about the statement structures and other results of parsing. Many of the fields in this structure are used only by the parser as it parses the user's input. Fields that are not described are reserved for use by the parser and are of no use to your procedure.

The fields relevant to procedure writing are described in detail below:

`head`
: gives the address of the procedure's statement structure. This address is completed by the call to SAS_XSPARSE. All other statement structures can be accessed after the first one is located by using the pointers in the `next` field of the statement structure. You can obtain the value of a `next` pointer by calling the SFNEXT routine, described in detail in Chapter 24, "Miscellaneous Routines and Values."

`last`
: indicates the last statement structure in the chain of structures created by your procedure.

`error`
: is the flag that is set when errors are discovered by the parser. Your procedure should check this field immediately after calling the parser to determine if any errors were detected. In this example, the procedure stops processing if any parsing errors occurred.

```
SAS_XSPARSE(grammar-function, NULL, &proc);
if (proc.error >= XEXITPROBLEM)
   SAS_XEXIT(XEXITSYNTAX,0);
```

`name`
: contains the first eight characters in the name of the procedure.

`lookfile`
: identifies the default lookup data set. The lookup data set is also the `curfile` at the beginning of your procedure. This field is set by the semantic actions @DS, @DSDFLT, and @SETDS.

STMT

Statement Structure

Declaration

```
typedef struct STMT *stmtptr;

/*--statement structure-------------------------------------------*/
struct STMT
 { stmtptr       next;      /* to next statment                 */
   char8         name;      /* statement name                   */
   char          opt[8];    /* statement options               */
   short         nfld;      /* number of fields                 */
   struct STLIST *fld;      /* statement fields                 */
   };

   struct FIELDSTR {
   short      n;            /* number of list elements used     */
   short      k;            /* number of list elements allocated */
   union {                  /* field value                      */
      char      *cval;      /* character                        */
      short     *ilist;     /* integer list                     */
      double    *flist;     /* float list                       */
      char      (*clist)[17]; /* character list                  */
      char8     *nlist;     /* name list                        */
      ptr       *efptr;     /* effect pointer list              */
      char8     *screen;    /* screen specification             */
      } u; } ;

/*--statement field structure-------------------------------------*/
struct STLIST
 { char        type;        /* field type                       */
   char        mode;        /* field mode                       */
   union {
      struct   FIELDSTR f;
      double   fval;        /* float                            */
      char8    nval;        /* name                             */
      ptr      fid;         /* file pointer                     */
      ptr      tlist;       /* triple list for dolists          */
     } v;
    };
```

Usage

1. It is not necessary to allocate or declare a STMTSTR because the needed ones have been allocated by the parser.

2. The `proc` structure, which is allocated and completed by the call to SAS_XSPARSE, contains the address of the first statement structure.

STMT *continued*

3. Use `proc.head` to access the statement structure of the PROC statement of your procedure.
4. Use the SF routines described in Chapter 24 to obtain information from the statement structure.

Description

When the statements that invoke a procedure are parsed, the information specified by the user in the PROC statement (and the other statements that can be used with the procedure) is stored in a series of statement structures.

Note: The statement structure is described in detail in this section to enable you to understand how information is passed from the user's statements through the parser to your procedure. You do not need to directly access the statement structure at any time. (In fact, in some languages, you cannot access the statement structure.) Use the SF routines to extract information from the statement structures. The SF routines can be called portably and greatly simplify the process of accessing information in the statement structures.

The types of data stored in the statement structures fall into these general categories:

- options
- parameters
- data set references
- lists of values or variables
- statement labels
- list counts
- user-defined input.

Your grammar must contain at least one @STMTINIT semantic action, which creates a statement structure to store the options, parameters, and data set names specified in the PROC statement. (This production in the grammar must also have the @PROCINIT semantic action to indicate it is the main statement structure for the procedure.) In general, the procedure's statement structure also contains the lists of variables for statements like the VAR statement or the BY statement. List statements require special coding to access the data stored in the statement structure. Refer to "Lists in the Statement Structure" later in this chapter for detailed information on list processing.

Statements other than list statements usually have a separate statement structure to store information. Each time one of these additional structures is created, which occurs when the @STMTINIT action is encountered in the grammar, it is chained to the last statement structure. The structures are created and chained together in the order that the user invokes the statements.

Relationship of the Statement Structure to the Grammar

Excerpts from a fictional procedure's grammar illustrate how the statement structures are defined for the procedure.

```
%include "stub.grm".

#--------statements--------------------------------------------------

PROGRAM =
   ANYSTMT ENDJB,

ANYSTMT =
   FICTIONSTMT  |
   BYSTMT       |
   SUMSTMT      |
   VARSTMT      ,

#--------statement productions---------------------------------------

FICTIONSTMT =                      ❶ @PROCINIT
                                   ❷ @STMTINIT( 24 )
   "FICTION"   FICTIONOPT*           @DSDFLT( 19, 1, 0, 4, 3, 4, 0 )
                                     @STMTEND ,

SUMSTMT =                          ❹ @STMTINIT( 2 )
                                     @STMTLIST( 1, 1 )
   "SUM"                             VARLIST
   <"/" SMOPT*>                      @STMTEND ,

VARSTMT =                          ❸ @STMTPROC
                                     @STMTLIST( 4, 3 )
   ( "VAR" | "VARIABLE$" |
     "VARIABLES$" )                  VARLIST ,

additional grammar statements to define
the options used in these statements
```

In this excerpt from the FICTION procedure's grammar, the @PROCINIT semantic action (❶) signals the parser that the statement structure created by the @STMTINIT semantic action (❷) is the main statement structure for the procedure. The VAR statement uses the @STMTPROC action (❸) to indicate that the VAR list is stored in the procedure's main statement structure. The BY statement does not need to be defined because the stub grammar defines a standard BY statement. The BY list is stored in the procedure's statement structure. The SUM statement creates its own statement structure by using the @STMTINIT semantic action (❹).

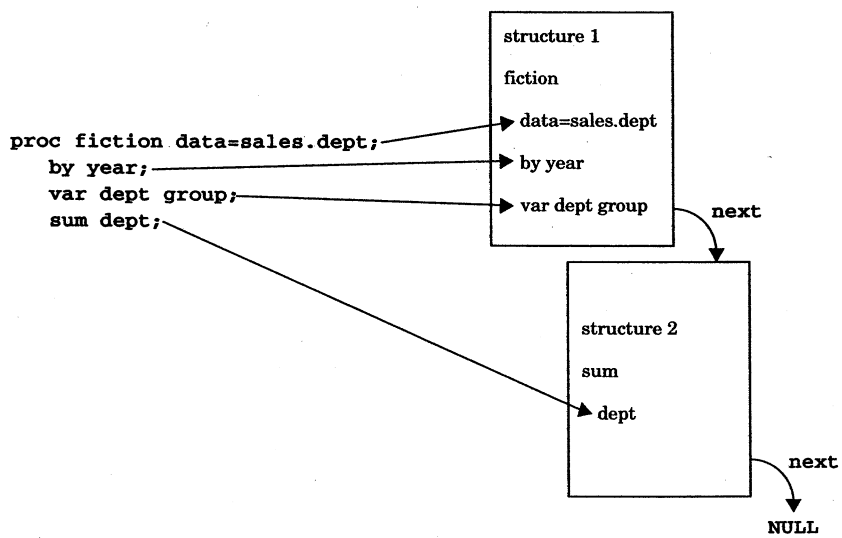
When the user enters these SAS statements, two statement structures are created and chained together.

STMT *continued*

```
proc fiction  data=sales.dept;
   by year;
   var dept group;
   sum dept;
```

First, a statement structure is created for the PROC FICTION statement. The name of the data set and the list of variables from the VAR and BY statements are stored in this first statement structure. When the SUM statement is parsed, a second statement structure is created, and the variables, options, and parameters specified in this statement are stored in it. This second structure is chained to the procedure's statement structure by placing the address of the second statement in the `next` field of the first structure. Each statement structure uses its own `next` field to store the location of the next structure. The last `next` field is null. Figure 20.1 illustrates this process.

Figure 20.1
Chained Statement Structures



Note: You are not required to create statement structures as described here. You can, if you prefer, use a single statement structure for your entire procedure. If you choose to do this, you are limited to 64 options for all statements in the procedure, and you must be sure to assign unique numbers for all lists, parameters, formats, and data sets. If you use this method, each statement to be stored in the procedure's statement structure must have the @STMTPROC semantic action.

Another alternative to the method described earlier is to begin a new statement structure for every statement regardless of whether it has options and parameters or is only a simple list statement. This method requires that each statement have the @STMTINIT semantic action. Each time the parser encounters the @STMTINIT semantic action in the grammar for your procedure, it creates another statement structure and links it to the previous ones. Since the procedure's statement structure also has the @PROCINIT action associated with it, the parser knows which statement structure should be first in the list of structures. The head of this linked list (the address of the procedure's statement structure) is stored in the `head` field of the parsing structure (PROC).

The call to SAS_XSPARSE establishes the location of the first statement structure. To access the other statement structures that are chained to the first one, use the SFNEXT routine to obtain the value of the `next` pointer. When the SFNEXT routine returns a NULL pointer, you have reached the end of the chain of statement structures. Refer to Chapter 24 for a detailed description of the SFNEXT routine.

Table 20.1 summarizes the relationship of the contents of the statement structure and the semantic actions in your procedure's grammar. Following Table 20.1 are detailed discussions of each of the fields mentioned in this table.

Table 20.1 *Relationship of Statement Structure to Semantic Actions*

Structure	Field Name (in order within structure)	Semantic Action	Contents of Field after Semantic Action Is Processed
STMT	`next`	@STMTINIT and @STMTEND	the address of the next statement structure. If there is only one statement structure, or for the last structure in a chain, this field is NULL.
	`name`	@STMTINIT	the name of the statement that has the @STMTINIT associated with it.
	`opt[8]`	@OPT(*n*)	the *n*th bit of the 64 possible bits in this field is set to 1.
	`nfld`	@STMTINIT(*n*)	*n*, the number of statement lists to allocate.
STLIST	`type`	@DS or @DSDFLT	4, indicating a SAS data set reference.
		@LABEL	2, indicating a parameter.
		@LIST	3, indicating a list.
		@LISTCT	3, indicating a list.
		@LISTVAL	3, indicating a list.
		@PARM	2, indicating a parameter.
		@PARMVAL	2, indicating a parameter.
		@STMTLIST	3, indicating a list.
	`mode`	@DS, @DSDFLT, @LABEL, @LIST, @LISTCT, @PARM, @PARMVAL, @STMTLIST, (also the CHAR production)	varies depending on the mode used with the semantic action. Refer to Table 20.2 for information on how the grammar mode corresponds to the mode field in the statement list.

(continued)

STMT *continued*

Table 20.1 *(continued)*

Structure	Field Name (in order within structure)	Semantic Action	Contents of Field after Semantic Action Is Processed
STLIST (*continued*)	`fval`	@PARM(*n*,1)	contains the numeric parameter value specified by the user.
		@PARMVAL(*n value*)	contains the value of the second argument to @PARMVAL.
	`nval`	@PARM(*n*,2)	contains the SAS name value specified by the user.
		@LABEL	contains the first 8 bytes of the label if the user specifies a label for the statement.
		@SASNAM	contains the libref or member name for the SAS data set currently being processed.
	`fid`	@DS	contains the fileid to a successfully opened SAS data set.
	`tlist`		reserved.
FIELDSTR	`n`	@STMTLIST, @LIST	contains the number of elements in the list.
	`k`		reserved.
	`cval`	@PARM(*n*,3) or @PARM(*n*,5)	contains a long character string or quoted string. Both uses of @PARM produce the same effect.
		@LIST	contains a long character string built by @LIST with no null terminators embedded until the end of the string.
	`ilist`	@STMTLIST used with VARLIST	for variable lists, contains the address of the list of variable numbers.
		@LISTCT	contains the address of an integer that represents the number of elements in a list.
		@MARK	contains the address of a list to which a negative number was added by this action.

(continued)

Table 20.1 (*continued*)

Structure	Field Name (in order within structure)	Semantic Action	Contents of Field after Semantic Action Is Processed
FIELDSTR (*continued*)	`flist`	@LIST	contains the address of a list of floating-point values built by the @LIST semantic action.
		@LISTVAL	contains the address of a list to which the second argument to @LISTVAL was added.
	`clist[17]`		reserved.
	`nlist`	@LIST	contains the address of a list of blank-padded eight-character names built by the @LIST semantic action.
	`efptr`		reserved.
	`screen`		reserved.

Parts of the Statement Structure

The statement structure contains two other structures. The main part of STMT describes the format of each statement structure created by the parsing process. The `fld` array of STLIST structures is usually where the parsing data are stored.

The STLIST structure, the last structure in the declaration, describes all of the possible forms for the data in the statement structure. One of the types of data described by the STLIST structure is the FIELDSTR structure, which contains information about lists.

The three sections of the statement structure are described separately in the following sections.

The STMT Structure

This part of the statement structure describes the statement and points to the next statement structure.

```
struct STMT
 { stmtptr        next;      /* to next statment                    */
   char8          name;      /* statement name                      */
   char           opt[8];    /* statement options                   */
   short          nfld;      /* number of fields                    */
   struct STLIST  *fld;      /* statement fields                    */
   };
```

The fields of the STMT structure are described in detail below:

`next` contains the address of the next statement structure in the chain of structures. For a single statement structure or the last structure in a chain, this field contains a null pointer.

STMT *continued*

`name` contains the name of the statement stored in this statement structure. The name stored here for the procedure statement is the procedure name, not PROC. If your grammar generates only one statement structure, this field is not needed by your program. If your grammar generates several statement structures, as in the PROC FICTION example given earlier, you need to access this field to know what statement the structure contains. The procedure's statement structure is always first, but additional structures are created in the order in which the user specifies the statements. The name of the statement structure is truncated to 8 bytes if necessary.

`opt[8]` provides 64 bits to represent on/off options that can be specified in the statement. The option numbers are defined in the grammar with the @OPT semantic action; the argument for the semantic action assigns the option to a specific bit. For example, the following assigns bit 2 to the NOPRINT option:

```
"NOPRINT"      @OPT(2)
```

If the user specifies the NOPRINT option when invoking the procedure, bit 2 is turned on (set to '1'B). The default value for all options is '0'B.

`nfld` contains the number of STLIST structures that are allocated for this statement. The `nfld` value is set by the @STMTINIT semantic action coded in the grammar, which must be at least as large as the largest number used as the first argument for any @DS, @DSDFLT, @LABEL, @LIST, @PARM, @PARMVAL, or @STMTLIST semantic action.

`fld` defines an array of STLIST structures. The number of array elements is determined by the value coded in the @STMTINIT semantic action of the grammar. Each element of this array describes a single parameter, data set reference, list of variables or values, format, label, or parameter value that can be used in the statement.

The STLIST Structure

This portion of the statement structure describes the data stored in the structure.

```
struct STLIST
 { char      type;        /* field type                    */
   char      mode;        /* field mode                    */
   union {
      struct    FIELDSTR f;
      double    fval;        /* float                         */
      char8     nval;        /* name                          */
      ptr       fid;         /* file pointer                  */
      ptr       tlist;
    } v;
 };
```

The fields of the STLIST structure are described in detail below:

`type` indicates the type of information stored in this element. Table 20.2 shows the possible values of `type`.

You can determine whether the user specified the parameter, list, or data set associated with this STMTFLD by checking to see if `type` is not equal to 0. The entire statement structure is set to 0s when allocated. Since there is no valid `type` of 0, when `type` equals 0, it indicates that this list, parameter, format, or data set was not specified. Therefore, you can tell the difference between a missing parameter (which has a `type` of 0) and a parameter that is explicitly set to 0 (which has a non-zero value for `type`).

`mode` describes in more detail the kind of information stored in this element of the array. The value in this field depends upon both the type of semantic action and the arguments for the semantic action. That is, for each of the `types`, the `mode` has several different meanings. Table 20.2 shows the values in the `mode` for each semantic action.

`v` is a union that contains the data stored in this statement structure. Parameter values and lists are stored in this field. The contents of `v` depends upon the values of `type` and `mode`. The possible contents of the union are described below.

- `f` is a structure that describes a list of values. See the description of the FIELDSTR later in this section for more information on the contents of this field.
- `fval` contains a floating-point number.
- `nval` contains an 8-byte character string.
- `fid` points to the fileid for an open file.
- `tlist` is a reserved field.

Table 20.2 illustrates what values are stored in the `type` and `mode` fields for specific semantic actions. Note that the values for the `mode` have different meanings depending on the value for the `type`.

Note also that the `mode` does not always match the argument in the corresponding semantic action. For example, @STMTLIST(n,1) refers to a `mode` of 11.

STMT *continued*

The FIELDSTR Structure

This part of the statement structure describes a list when that is the type of information stored in one of the STLIST elements. For more information on how lists are stored, refer to "Lists in the Statement Structure," later in this chapter.

```
struct FIELDSTR {
short      n;              /* number of list elements used      */
short      k;
union {                    /* field value                        */
   char      *cval;        /* character                          */
   short     *ilist;       /* integer list                       */
   double    *flist;       /* float list                         */
   char      (*clist)[17];
   char8     *nlist;       /* name list                          */
   ptr       *efptr;
   char8     *screen;
   } u; } ;
```

The fields of the FIELDSTR are explained below:

n is the number of items in the list.

k is a reserved field.

u is a union that can contain the following types of data:

`cval` is an array of characters. Use this description when the user specifies

- □ a character value for a parameter argument, @PARM(*n*,5)
- □ a list of extra long character strings, @LIST(*field*,5)
- □ a statement label, @LABEL.

`ilist` is an array of integers. This description is used for the list of variable numbers corresponding to the variables named in the VAR statement or any statement that lists variable names. Use this description when the semantic action is @LISTCT, @MARK, or @STMTLIST(*field,n*), where *n* is 1, 2, or 3. The actual values for the list are not stored here; only the variable number is stored in this list. You must use the variable numbers in calls to SAS_XVNAME to obtain more information about the variable. This is where VARLIST and ONEVAR lists are stored.

`flist` is an array of floating-point numbers. This is where expanded numeric lists are stored (NUMLISTs with mode=1).

`nlist` is an array of 8-byte character strings. This is where NAMELIST and ONENAME strings are stored.

Accessing Data in the Statement Structures

Once the information parsed by your grammar is stored in the statement structures, you need to be able to access this information in your procedure to determine what statements, variables, options, and other grammar elements the user specified when invoking the procedure. To do this, use the SF routines described in Chapter 24. Table 20.2 summarizes the information stored in the statement structures by various elements of the grammar, shows the type and mode of each type of data, and then lists the SF routine that accesses the information in the statement structure.

Table 20.2 *Storing and Retrieving Information in the Statement Structures*

Semantic Action	type	mode	Type of Data Stored	Routine for Extracting Data
@PARM(*n*,1) or @PARMVAL	2	1	numeric value	SFF or SFFD
@PARM(*n*,2) or @LABEL	2	2	8-byte character value	SFC8 or SFC8D
@PARM(*n*,3) or @PARM(*n*,5)	2	3 or 5	extra long character value	SFN (length), SFLP (address)
@STMTLIST(*n*,1)	3	11	list of numeric variables	SFLP
@STMTLIST(*n*,2)	3	12	list of character variables	SFLP
@STMTLIST(*n*,3)	3	13	list of mixed variables	SFLP
@STMTLIST(*n*,4)	3	1	list of numeric values	SFLP
@STMTLIST(*n*,7)	3	2	list of 8-byte character values	SFLP
@LIST(*n*,1)	3	1	list of numeric values	SFLP
@LIST(*n*,2)	3	2	list of 8-byte character values	SFLP
@LIST(*n*,5)	3	5	list of long character values, each with a null terminator	SFLP (address of first string), SAS_ZPARMN (number of strings), SAS_ZPARM (address of a specific string)
@DS(*n*,1,...)	4	1	input data set	SFFILE
@DS(*n*,2,...)	4	2	output data set	SFFILE

(*continued*)

STMT *continued*

Table 20.2 (continued)

Semantic Action	`type`	`mode`	Type of Data Stored	Routine for Extracting Data
@DS(*n*,4,...)	4	1 or 2	input or output, depending on how the data set was successfully opened	SFFILE
@DS(*n*,6,...)	4	1 or −1	input, if the attempt to open is unsuccessful, do not treat this as an error	SFFILE

Lists in the Statement Structure

The parsing process creates several types of lists to store the information that is passed to your procedure. Separate lists are created for

- all variables in the data set
- the variables specified in each statement, such as the VAR or BY statement
- the values specified for special statements, such as the VAXIS statement of the PLOT procedure.

Lists are created by using the @STMTLIST or @LIST semantic actions in the grammar. See the "Standard Semantic Actions" section of Chapter 21, "Grammar Language," for more details on the @LIST and @STMTLIST actions. The following sections describe the kinds of lists that can be stored in the statement structures.

Data Set Variable List

The first type of list is created when the parser opens a SAS data set for input and obtains access to the variables and the descriptive information in the data set. Although no physical list of data is created, the information available about these variables is called the data set variable list. The data set variable list is the relative position of each variable in the data set. Conceptually, for a SAS data set with six variables, the data set variable list looks like the following:

Variable	Data Set Variable List Number
DATE	1
TEMP	2
HUMIDITY	3
RAINFALL	4
BAROMETR	5
WINDSPED	6

When any other list of variables is created, the variable names entered by the user are compared to the information in the data set variable list, and the relative position number of the variable is stored in the variable list instead of the name of the variable. This is described in more detail in the following section.

Variable Lists for Statements

The second type of list is created by the parsing process when the @STMTLIST action is used with a second argument of 1, 2, or 3. (The first argument simply specifies the list number.) This kind of @STMTLIST action is used for a statement like the VAR statement, which requires a list of variables from an existing SAS data set. In most cases, your procedure accesses lists of variables by calling SAS_X routines, such as SAS_XVLIST, and specifies the list number as one of the arguments to the routine.

As described for the data set variable list, the lists created for these statements are actually lists of integers where the relative positions of the variables are stored. For example, suppose you code this statement in your grammar:

```
VARSTMT = @STMTPROC @STMTLIST(1,3) "VAR" VARLIST,
```

When the user specifies the following VAR statement for the data set used in the previous example, the list for the VAR statement is LIST 1, as specified by the first argument in the @STMTLIST semantic action.

```
var  temp rainfall date;
```

For this example, LIST 1 contains the values 2, 4, and 1, which are the relative positions of these variables in the data set variable list. The parsing process also stores information that describes LIST 1 in the first FIELDSTR structure in the statement structure.

Use the SFLP routine to access the pointer to a variable list. Then pass the pointer returned by SFLP to the SAS_XV routines to access the variables.

Note: The only list that has a standard number is the BY variable list, which is always assigned to LIST 0. Therefore, for procedures that permit a BY statement, you can use 0 as the list number in the call to SFLP.

Value Lists

Value lists, the third kind created by the parsing process, are created when the @STMTLIST action is coded with a second argument of 4 or 7, or when you use the @LIST action. For value lists, the actual values are stored in the statement structure. If you code this statement in your grammar

```
VALUESTMT = @STMTPROC @STMTLIST(2,4) "VALUES" NUMLIST,
```

and the user specifies this VALUE statement,

```
VALUE  10 20 30 40 50;
```

the list for the VALUE statement is LIST 2, and it contains these values: 10 20 30 40 50. Information concerning LIST 2 is stored in the first FIELDSTR structure in the statement structure.

The value list is the list most commonly accessed directly by your procedure. To access the values in this type of list, use the SFLP routine to obtain the pointer to the list. Then dereference the pointer to access the values in the list.

WZ_ENV

Working Environment Structure

Declaration

```
struct   WZ_ENV {              /* structure describing the environment */
   char   *model_name;         /* machine model name                   */
   char   *model_num;          /* machine model number                 */
   char   *serial;             /* machine serial number                */
   char8  os_name;             /* operating system name                */
   char8  os_family;           /* operating system family              */
   char16 os_num;              /* operating system number              */
   char16 jobid;               /* job or process id                    */
   char16 sup_ver_long;        /* SAS supervisor version               */
};
```

Usage

1. Allocate or declare a copy of the WZ_ENV.
2. Call SAS_ZSYSINF and pass the address of the WZ_ENV structure you allocate or declare.

Description

The WZ_ENV structure is used by the SAS_ZSYSINF routine to pass a description of the host system that is currently running the procedure. The fields of the WZ_ENV structure are described below.

`model_name`
: points to the machine model name, such as VAX or IBM.

`model_num`
: points to the machine model number, such as 11/780.

`serial`
: points to the machine serial number.

`os_name`
: is the specific name of the operating system, such as MVS/XA.

`os_family`
: is the generic name of the operating system family, such as OS.

`os_num`
: is the operating system number.

`jobid`
: is the name of the job or process running the procedure.

`sup_ver_long`
: contains a 16-character string. The first 4 bytes of this string contain the current release of the SAS System (for example, 6.07). The remaining bytes are for internal use.

Note: Some systems can have more than one current CPU, such as the IBM 3090 series. For this reason, the three fields that contain CPU information (`model_name`, `model_num`, and `serial`) are actually pointers to strings that can contain several values. Each value is null-terminated. For example, if you are running on an IBM 3090 with serial numbers 123456, 223456, and 323456, `model_name`, `model_num`, and `serial` point to the following strings respectively:

```
IBM\0IBM\0IBM\0\0
3090\03090\03090\0\0
123456\0223456\0323456\0\0
```

XCOPSTR

Open Utility File Parameter Structure

Declaration

```
struct XCOPSTR
{
    fname3  fn;      /* name of the file to be opened          */
    int     mode;    /* open mode and access pattern            */
    int     max_rc;  /* 0 or 1                                  */
                     /* 0 if application is NOT checking return */
                     /*      codes from xc calls                */
                     /* 1 if application is checking return codes */
                     /* NOTE: if max_rc is set to 0 the task will */
                     /*       terminate if xc encounters an error */
                     /*       level return code.                */
    long    growth[4];

};
```

Usage

1. Allocate or declare a copy of the XCOPSTR.
2. Complete the `fn`, `mode`, and `max_rc` fields.
3. Call SAS_XCOPEN to open the file using the information in the XCOPSTR.

Description

The open file parameter structure (XCOPSTR) is used by the SAS_XCOPEN routine. The open utility file parameter structure communicates information about how to open a utility file. A utility file is a type of SAS file that can be used for temporary storage by your procedure. The XCOPSTR is described here. Fields that are not described are reserved for use by SAS/TOOLKIT software.

XCOPSTR *continued*

`fn`
: is the complete filename for the SAS file. The `fname3` type is defined as follows:

```
typedef  struct   {  char8     libname;
                     char8     memname;
                     char8     memtype;
                     ptr       lib_handle;
                  }
         fname3;
```

The parts of the `fn` field are as follows:

`libname`
: specifies the libref for the SAS data library containing the file. Leave this blank unless you require a specific reference name for your procedure. If this field is blank, the default library is used, which is usually WORK.

`memname`
: names the file. If this field is blank when you call SAS_XCOPEN, a member name is created for you, but it is better to specify one of your choice. The member name cannot have more than six characters.

`memtype`
: indicates the member type. Specify UTILITY, in uppercase.

`lib_handle`
: reserved.

`mode`
: specifies the mode of opening and the access flag. You must add the mode and the access flag together and assign that value to this field. The modes of opening are as follows:

XC_INPUT
: read-only access.

XC_OUTPUT
: write access. Creates a new utility file and replaces any existing one.

XC_UPDATE
: read and write access to an existing utility file.

The access flags are as follows:

XC_RAND
: indicates random access.

XC_SEQ
: indicates sequential access.

`maxrc` is a flag indicating whether your procedure will perform error checking after all I/O calls to the data set. If you set this field to 0, the system performs all error checking for you. In this case, when an I/O error occurs, the system automatically terminates your procedure and prints the appropriate error messages. In many cases, however, you may prefer to have your procedure test the return code and decide how to handle error conditions. In this case, set this value to 1.

`growth` reserved.

XOPNSTR

Open File Parameter Structure

Declaration

```
struct XOPNSTR {
   char     libname[8];
   char     memname[8];
   int      opnmode;
   int      maxrc      ;        /* maximum return code alowable  */
   char    *type;
   int      typelen;
   char    *label;
   int      labellen;
};
```

Usage

1. Allocate or declare a copy of the XOPNSTR.
2. Complete the `libname`, `memname`, `opnmode`, and `max_rc` fields. If you are opening the SAS data set for output, you may also want to fill in the `type`, `typelen`, `label`, and `labellen` fields.
3. Call SAS_XOOPEN to open the SAS data set using the information in the XOPNSTR.

Description

The open file parameter structure (XOPNSTR) is used by the SAS_XOOPEN routine. The open file parameter structure communicates information about a SAS data set when it is opened. The fields of the XOPNSTR are described here.

`libname` specifies the libref for the SAS data library containing the data set. Leave this blank unless you require a specific reference name for your procedure. If this field is blank, the USER libref is used. Refer to the description of the USER= option in Chapter 16, "SAS

XOPNSTR *(libname continued)*

System Options," of *SAS Language: Reference, Version 6, First Edition,* for more information on the USER libref.

`memname` specifies the member name of the SAS data set. If this field is blank when you call SAS_XOOPEN, one of the default data set references is used. The default used depends on the value specified for the `opnmode` field. _LAST_ is used for input or update mode data sets; _DATA_ is used for output mode data sets.

`opnmode` specifies the mode of opening and the access flag. You must add the mode and the access flag together and assign that value to this field. The modes of opening are as follows:

XO_INPUT	read-only access.
XO_OUTPUT	write access. Creates a new SAS data set.

The access flags are as follows:

XO_BYRWND	indicates that BY groups will be read more than once.
XO_RAND	indicates random access.
XO_SEQ	indicates sequential access.
XO_2PASS	indicates that the entire SAS data set will be read more than once.

`maxrc` is a flag indicating whether your procedure will perform error checking after all I/O calls to the data set. If you set this field to 0, the system performs all error checking for you. In this case, when an I/O error occurs, the system automatically terminates your procedure and prints appropriate error message. In many cases, however, you may prefer to have your procedure test the return code and decide how to handle error conditions. In this case, set this value to 1.

Note: If you are using BY-group processing on the data set, you must set this code to 0.

`type` specifies the special data set type. This value is either blank or one of the valid values for the TYPE= data set option, which is illustrated here.

```
PROC FACTOR DATA=OLD(TYPE=CORR);
```

`typelen` is the length of the character string containing the special data set type.

`label` specifies the SAS data set label. This is the value the user specifies for the LABEL= data set option, which is illustrated below.

```
PROC FACTOR DATA=OLD(LABEL='SALES DATA FOR 1984');
```

`labellen` is the length of the character string containing the SAS data set label.

Chapter 21 Grammar Language

Introduction

The main elements used in coding a grammar are

- symbols and operators
- lexicals
- standard productions
- semantic actions.

This chapter describes in detail these types of grammar elements. Note that in addition to these elements, the grammar also uses terminals, but these are simply keywords chosen by the programmer as mnemonic terms for invoking statements and options.

Symbols and Operators

Table 21.1 lists the symbols and operators available to your grammar and then illustrates how to use them. In several cases, the table illustrates the use of an operator by nesting a reference to another production in the production that uses the operator.

Table 21.1
Grammar Symbols and Operators

<table>
<tr><th>Symbol</th><th>Sample Use</th><th>Description</th></tr>
<tr><td>no symbol (juxtaposed)</td><td><code>"DATA" "="</code></td><td>Both elements are required in the order specified.</td></tr>
<tr><td>|</td><td><code>"VAR"|"VARIABLES"</code></td><td>Either element is valid but not both.</td></tr>
<tr><td>*</td><td><code>GENSTMT = GENSTMTOPT*</code>
<code>GENSTMTOPT = OPTION1|</code>
<code>OPTION2|</code>
<code>OPTION3,</code></td><td>Zero or more occurrences of the item preceding the asterisk are permitted. In this example, the user can specify any, all, or none of the options defined by GENSTMTOPT or repeat any option.</td></tr>
<tr><td>+</td><td><code>GENSTMT = GENSTMTOPT+</code>
<code>GENSTMTOPT = OPTION1|</code>
<code>OPTION2|</code>
<code>OPTION3,</code></td><td>One or more occurrences of the item preceding the plus sign are permitted. In this example, the user must specify at least one of the options defined by GENSTMTOPT.</td></tr>
<tr><td>;</td><td><code>GENSTMT = GENSTMTOPT;'-'</code>
<code>GENSTMTOPT = OPTION1|</code>
<code>OPTION2|</code>
<code>OPTION3,</code></td><td>The first item can be specified any number of times, but each occurrence must be separated from the next by the second item. In this example, the value of GENSTMTOPT can be any of the three options, but the user must separate specified options with a hyphen, for example OPTION1-OPTION3-OPTION2.</td></tr>
<tr><td>( )</td><td><code>GENSTMTOPT = (OPTION1|</code>
<code>OPTION2)</code>
<code>OPTION3,</code></td><td>Groups items to clarify the meaning. In this example, the user must specify one of the following:
OPTION1 and OPTION3
OPTION2 and OPTION3.</td></tr>
<tr><td><></td><td><code>GENSTMTOPT = <OPTION1></code>
<code>OPTION2</code></td><td>Items within the brackets are optional. In this example, the user can specify either of the following:
OPTION1 and OPTION2
OPTION2.</td></tr>
</table>

Note: The order of precedence for the operators is listed here: +, *, juxtaposition, |, and ;. This order can be changed by the ( ) (parentheses) and <> (angle brackets) grouping symbols.

Lexicals

The following portions of the grammar are lexicals, which describe the general class of the token that is required. Lexicals are most commonly used to identify what type of value is expected for a parameter value, although there are other uses for lexicals as noted below. For detailed information on the semantic actions used in the examples, refer to "Semantic Actions" later in this chapter.

COMPOUND

The COMPOUND lexical indicates a compound of names that are joined by periods. Use this lexical for *libname.memname* or LAST.*by-variable* constructions.
The following example illustrates this lexical:

```
(("LIBRARY" | "LIB") "=" @PARM(1,5) (NAME|COMPOUND))
```

DATE

The DATE lexical indicates that the input must be a date literal in the DATE*w*. format (for example 01MAY91). The date value entered for this lexical is stored as an integer that represents the offset in days from 01JAN60. Dates before 01JAN60 are stored as negative integers.

DATETIME

The DATETIME lexical indicates that the input must be a datetime literal of the form '01MAY91:12:34:56'dt. The datetime value entered for this lexical is stored as an integer that represents the offset from midnight 01JAN60. Datetime values before midnight 01JAN60 are stored as negative integers.

ENDJB

The ENDJB lexical indicates the end of a statement. You use it only once in your grammar in the PROGRAM production that defines all of your statements.
The following example illustrates this lexical:

```
PROGRAM = ANYSTMT ENDJB,
```

INT

The INT lexical indicates that the input must be an integer. A negative sign preceding the integer is permitted.
The following example illustrates this lexical:

```
"DECIMALS" "=" @PARM(3,1) INT
```

MISSING

The MISSING lexical indicates that the input must be a missing value, which can be any of the following values:

.

._

.A through .Z

Keep in mind that you would probably use this lexical in combination with another lexical to indicate that the input can be of the type specified by the other lexical or a missing value. Using just this lexical alone permits only missing values for the parameter.

The following example illustrates this lexical:

```
"NUMVALUE" "=" @PARM(5,5,80) (NUMBER|MISSING)
```

NAME

The NAME lexical indicates that the input must be an unquoted character string of 200 characters or less that begins with a letter or underscore and contains only letters, digits, or underscores.

The following example illustrates this lexical:

```
"PREFIX" "="    @PARM(4,2) NAME
```

NBR

The NBR lexical indicates that the input must be a number containing a decimal point or exponent, such as 9.9 or 9.9E9. A negative sign preceding the number is permitted. Missing values are not accepted.

The following example illustrates this lexical:

```
"FUZZ" "=" @PARM(3,1) NBR
```

QS

The QS lexical indicates that the input must be a quoted string. The maximum length for a quoted string is 200.

The following example illustrates this lexical:

```
"TITLE" "="    @PARM(3,5) QS
```

TIME

The TIME lexical indicates that the input must be a time literal of the form '12:34:56't. The time value entered for this lexical is stored as an integer that represents the offset from midnight, which is 0. Time values are always positive.

Standard Productions

The following descriptions are for the standardized productions that are part of the stub grammar. These productions can be referenced within productions you define.

BYSTMT

The BYSTMT production allows a BY statement to be used with your procedure. A BY statement permits variable lists of both character and numeric type variables. Each list item can be preceded by the keyword DESCENDING, and the entire list can be followed by the keyword NOTSORTED. If your procedure supports BY-group processing, use this production name as one of the available statements in your grammar's list of possible statements, but do not create a production in your grammar for the BY statement. Including the BYSTMT production completely defines the BY statement for your procedure.

DSFIELD

The DSFIELD production indicates a data set specification. The data set name can be any of the following:

- a two-level name in the format *libref.memname*
- a single-level name consisting of just a *memname*
- one of the automatic variable names _LAST_ or _DATA_.

In addition, you can specify data set options enclosed in parentheses. Use this production with the @DS semantic action.

NAMELIST

The NAMELIST production allows lists of SAS names or no list. Lists can be individual SAS names or single-hyphenated lists of SAS names or any combination of these, such as the following:

```
x
x y
x1-x5.
```

You cannot, however, use colon-format lists or double-hyphenated lists, such as the following:

```
x-numeric-a
x--a
ab:
```

Use this production in combination with the @STMTLIST semantic action to store the list of values.

The NAMELIST is useful for lists of names for output data sets. In most cases, however, when you use NAMELIST, you must also use one of the @SPECIAL semantic actions. Refer to the descriptions of @SPECIAL(1) and @SPECIAL(2).

NUMBER

The NUMBER production allows any floating-point number or integer, negative or unsigned, or any date, time, or datetime literal. Missing values are not permitted with this production.

NUMLIST

The NUMLIST production permits a list of values that can be specified in any of the following formats:

num-1 <, . . . *num-n*>
start TO *end*
start TO *end* BY *increment*

where *start* and *end* are integers or dates (in the format 'SASdate'D) and *increment* is an integer or one of the following:

DAY	MINUTE
DTDAY	MONTH
DTMONTH	QTR
DTQTR	SECOND
DTWEEK	WEEK
DTYEAR	YEAR
HOUR	

Using this production allows the user to specify many numeric values without having to list each value. Use this production after the @STMTLIST semantic action.

A NUMLIST can optionally be enclosed in parentheses, separated by commas, or both. The following are examples:

(1 2 3)
1 2 3
(1,2,3)
1,2,3
1 2 3 to 10 by 2, 7
(7 8 to 20 5 10 3 to 9 by 2).

Numeric values can also include signed numbers and noninteger values.

NUMORMISS

The NUMORMISS production allows anything included in the NUMBER production plus missing values.

NUMORMISSLIST

The NUMORMISSLIST production allows anything included in the NUMLIST production plus missing values. Use this production after the @STMTLIST semantic action.

ONENAME

The ONENAME production indicates that you allow only a single valid SAS name. You can also use ONENAME when you want an exact number of names by specifying the ONENAME production multiple times. Use this production after the @STMTLIST semantic action.

ONEVAR

The ONEVAR production indicates that you allow only a single variable. By specifying the ONEVAR production multiple times, you can get an exact number of variables. Use this production after the @STMTLIST semantic action.

VARLIST

The VARLIST production allows any acceptable variable list, including the following:

A B C
A-B
A--Z
A1-A10
A-NUMERIC-B
A-CHARACTER-B
CHARACTER
NUMERIC
ALL
A:

This production also accepts no variable list. When the user specifes variables, the production ensures that the variables exist on the current look-up data set.

Use this production with the @STMTLIST semantic action to store the list of variables.

Semantic Actions

The semantic actions are coded in the grammar in the following form:

@action<(arg-1<, . . . arg-20)>

These actions call routines in the parser to build statement structures and lists of variables and to perform validations and other specialized actions such as counting or marking. In some cases, where you place a semantic action in your grammar is important to ensure that the correct token is affected by the action. In most cases, you put the semantic action before the item to which it applies. The exceptions to this are the @DS and @DSDFLT semantic actions, which appear after the DSFIELD production, and the @OPT semantic, which can be specified after the terminal for the option. In general, the only grammar element that can come between the semantic action and the item it modifies is another semantic action.

If a semantic action requires a specific placement in your grammar, the description notes the requirement. These actions are also flagged with an asterisk (*) in the table below. The most commonly used actions are listed here.

Semantic Action	Description
@DS	opens a data set.
@DSDFLT	opens a default data set if data set reference is not given.
@DSLKUP	makes a data set the one currently being processed.
@LABEL	stores statement label in statement structure.
@LIST	adds value to a list.
@LISTCT	adds list count to another list.
@LISTVAL	adds a constant to a list.
@MARK	puts negative markers into variable lists.
@OPT	sets an option on.
@PARM	sets a parameter equal to an input value.
@PARMVAL	sets a parameter value to a constant.
@PROCINIT	initializes the semantic action processor.
@RANGE *	checks if input number is in valid range.
@RESETN	resets list count to 0.
@SASNAM	ensures that the token is a valid SAS name of up to three levels.
@SPECIAL *	alters normal parsing processes.
@STMTEND	finishes a statement structure.
@STMTINIT *	allocates a statement structure.
@STMTLIST	places list in a statement structure.
@STMTPROC	points back to procedure's statement structure.

Chapter 3, "Writing a Grammar," explains how semantic actions add data to the statement structures. In addition, Chapter 20, "Data Structures," provides a detailed description of the statement structure, STMT.

Standard Semantic Actions

The semantic actions typically needed for your grammar are described in the following section. Note that the *field* described for many of these actions refers to the relative position in the `fld` array where information is stored by the action. Specific actions do not require specific numbers.

@DS

The @DS semantic action permits the user to specify a data set name. Code this action as follows:

```
@DS(field,mode,status,access-1,access-2,access-3,errors<,process>)
```

where

field is the index to the `fld` array in the statement structure.

mode is one of the following:

1 indicates an input data set. The `mode` field of the STLIST data structure is set to 1.

2 indicates an output data set. The `mode` field of the STLIST data structure is set to 2.

4 indicates an attempt to open the data set for input; if it does not exist, open it for output. The `mode` field of the STLIST data structure is set to 1 for input or 2 for output. Procedures such as FSEDIT use this mode to indicate that if the screen data set exists, read it, and if it does not exist, create it.

6 indicates an attempt to open the data set for input; even if the attempt is unsuccessful, do not treat this condition as an error. If this condition occurs, the `mode` field of the STLIST data structure is set to 255: otherwise, it is set to 1.

status is a flag that indicates any of the following:

0 indicates no special status.

1 indicates that this should be the the current look-up data set. That means that all variables specified in the statements entered by the user refer to variables in this data set. The data set's fileid is stored in `proc.lookfile`.

access-1, access-2, and *access-3*
must be used together to indicate how to open the data set and what kind of access the procedure has to the data set. Before you specify these parameters, consider the following questions:

- □ Does your procedure need to have input or output access to the data set?
- □ Will your procedure read the data set in sequential order or random order?
- □ Does your procedure need to reread the data set or a BY group in the data set?
- □ Do you need to maintain control of the entire data set while your procedure processes it (called *member-level access*), or do you need access to only one record at a time (*record-level access*)? In most cases, record-level access is possible only if your procedure works with only one observation at a time and never rereads any portion of the data set.

Once you have determined the answers to these questions, you can use Table 21.2 to select the appropriate values for the three access parameters. The *access-1* and *access-2* parameters specify the level of access needed by your procedure. The values for these parameters are as follows:

2 member-level access

4 record-level access.

The values for *access-3* indicate the following modes of access:

1 random access

2 data set may be reread

3 BY groups may be reread

4 sequential access only.

Note: Keep in mind that the values you specify in the access parameters of the @DS semantic action limit the kinds of data sets that your procedure can process. If you specify a type of access that is not supported by the engine used to read a data set, your procedure terminates with an appropriate message. For example, if your procedure must reread a data set and you specify a value of 2 or 3 for *access-3*, when a user attempts to use your procedure with the sequential engine the procedure fails.

Table 21.2 *Values for Access Arguments of the @DS Semantic Action*

Access Description	*access-1*	*access-2*	*access-3*
input, sequential access, no rereading of the data set or BY groups	4	2	4
input, sequential access, reread data set (your procedure calls SAS_XBYRWND with a *rewind-type* of F)	2	2	2
input, sequential access, reread BY groups (your procedure calls SAS_XBYRWND with a *rewind-type* of B)	2	2	3
input, random access (your procedure calls SAS_XOPNT with specific record IDs)	2	2	1
output	2	2	4

errors sets a flag indicating if the procedure should automatically exit when an I/O error is encountered. The values are

0 automatically exit when errors occur.

1 do not exit. The procedure will check for errors and handle them.

process sets a flag indicating whether to perform attribute processing (which includes processing for ATTRIB, FORMAT, INFORMAT, LABEL, or LENGTH statements), or WHERE processing (which includes processing for the WHERE statement or data set option). This parameter is optional. Values are

0 performs both attribute and WHERE processing. This is the default action when you omit the parameter.

1 performs only attribute processing.

2 performs only WHERE processing.

3 does not perform any WHERE or attribute processing.

4 performs both attribute and WHERE processing, but gives no error messages for attribute errors.

5 performs only attribute processing, but gives no error messages for attribute errors.

Use the @DS action in conjunction with the DSFIELD standard production. The coding for a statement that processes an output data set is similar to the following:

```
OUTSTMT = ("OUT" "=" DSFIELD)  @DS(field,2,0,2,2,4,0,0) ,
```

The DSFIELD production performs actions to analyze the data set name.

@DSDFLT

The @DSDFLT semantic action supplies a default SAS data set if the user does not specify a data set reference. You code this action as follows:

```
@DSDFLT(field,mode,status,access-1,access-2,access-3,errors<,process>)
```

The arguments for this action are the same as those described for the @DS action.

Use the @DS action in conjunction with the @DS semantic action and the DSFIELD standard production. The coding for a statement that accesses a default output data set when the user omits a data set reference is similar to the following:

```
OUTSTMT = "OUTDS"  OUTOPTS*      @DSDFLT(field,2,0,2,2,4,0,0) ,
OUTOPTS = ("OUT" "=" DSFIELD)  @DS(field,2,0,2,2,4,0,0)
          | other options ,
```

The @DSDFLT semantic action is ignored if the user specifies a data set reference.

@DSLKUP

The @DSLKUP semantic action changes the look-up data set. Code this action as follows:

```
@DSKLUP(field)
```

where *field* is the position of the file in the procedure's statement structure. That is, the file referred to by *field* must be an open file, and the fileid for it must be in the procedure's statement structure.

The look-up data set is typically established when you specify a value of 1 for the third argument of the @DS or @DSDFLT semantic actions. Any time you create a statement that accepts variable lists, the list of variables is checked against the list of variables in the look-up data set. If you have a statement that lists variables that come from a secondary data set, not the current look-up data set, use the @DSKLUP semantic action to direct the system to look in the secondary data set to verify the variable list. Be sure to reset the look-up data set to the default after you process the statement. The following example shows how to change the look-up data set and then reset it to the default, which is *field* 1 in this example.

```
MYSTMT =      @STMTPROC
              @STMTLIST(4,3)
              @DSLKUP(2)
"MYSTMT"      VARLIST
              @DSLKUP(1)  ,
```

@LABEL

The @LABEL semantic action writes the first 8 bytes of the statement label into a statement field. Code this action as follows:

```
@LABEL(field)
```

where *field* is the index to the `fld` array.

Use this action if you want to keep track of the labels that the user assigns to a statement. For example, suppose the user codes statements with labels, like this:

```
STMT2: VAR  HEIGHT WEIGHT;
```

In this case, the character string STMT2 can be stored in the statement structure by using the @LABEL semantic action. The label is written into the FIELDSTR field with `type`=2 to indicate this value should be treated as a parameter and `mode`=2 to indicate this is an 8-byte name.

This example stores the label assigned to a statement in the fifth element of the `fld` array:

```
MYSTMT =    @STMTPROC
"MYSTMT"    @LABEL(5)
            @STMTLIST(4,3)
            VARLIST ,
```

@LIST

The @LIST semantic action creates a new list or adds an element to an existing list. The @LIST semantic action differs from @STMTLIST in the following ways:

- The @LIST actually writes a value to the list; @STMTLIST simply initializes the `mode` and `type` fields.
- @STMTLIST, in conjunction with a production such as VARLIST, builds lists of variable numbers as well as lists of values. @LIST cannot look up variable numbers in a data set, so this action cannot be used to build variable lists. It is used to create lists of values.

You use either @STMTLIST or @LIST to define a list in a grammar. Do not use both of the semantic actions together.

You code this action as follows:

```
@LIST(field<,mode<,max-length>>)
```

where

field is the index to the `fld` array.

mode specifies the type of data in the list. Valid values are the following:

1 indicates a list of doubles. This list is pointed to by the `flist` field of the FIELDSTR.

2 indicates a list of 8-byte character strings. If the item is a quoted string, quotes are removed before the item is stored in the list. This list is pointed to by the `clist` field of the FIELDSTR.

5 indicates a list of null-terminated character strings of varying lengths. If the item is a quoted string, quotes are removed before the item is stored in the list. This list is pointed to by the `cval` field of the FIELDSTR. The total length of all items in the length, including the null terminators, is stored in the `n` field of the FIELDSTR. Use SAS_ZPARM and SAS_ZPARMN to access the items in this kind of list. By default, no more than MACSHORT−1 characters (which does not include the null terminator) can be stored in this type of list, but you can further limit the length of an item. See *max-length*. Refer to Chapter 24, "Miscellaneous Routines and Values," for more information on the MACSHORT value.

max-length indicates the maximum number of characters per item that should be stored for lists with a mode of 5. This number does not include the null terminators for mode 5 lists.

@LISTCT

The @LISTCT semantic action adds to one list the count of the elements that are in another list. To use this action, code as follows:

```
@LISTCT(dest-field,source-field)
```

where

dest-field is the list receiving the count. This list must be a list of numeric values like the one created by the @STMTLIST(*field*,4) action. This list is pointed to by the `ilist` field of the FIELDSTR. In the STLIST structure, the `type` is set to 3 and the `mode` is set to 11.

source-field is the list that is being counted.

The @LISTCT semantic action allows you to keep track of how many items were specified by the user in a statement that allows varying numbers of elements.

For example, if you want to know how many variables the user specifies, code your VAR statement to include a count of the number of variable names:

```
VARSTMT = @STMTPROC  @STMTLIST(2,1)
          ("VAR" | "VARIABLES") VARLIST  @LISTCT(4,2) ,
```

In this example, list 2 contains the variables specified in the VAR statement, and list 4 contains the number of variables in list 2.

@LISTVAL

The @LISTVAL semantic action stores an integer as a double in a numeric list. Code the action as follows:

```
@LISTVAL(field,value)
```

where

field is the list number (that is, the index to the `fld` array in the statement structure) where the value should be stored. This list is pointed to by the `flist` field of the FIELDSTR.

value is the integer value to be stored in the list. The value is written to the list as a double.

@MARK

The @MARK semantic action stores a negative marker in a variable list. Code the action as follows:

```
@MARK(positive-integer)
```

where *positive-integer* is the number to be negated and inserted in the current list. You can use these negative integers as markers when the other values in the list either cannot be negative or cannot have the values you assign with the *positive-integer* argument. Use this action to indicate when variables should be crossed or grouped. For example, if your procedure permits crossing and grouping that can be represented in the following format, you need to indicate how the variables are paired and which pairs should be cross-tabulated:

```
A*(B C)
```

A convenient way to indicate these relationships is to define certain negative numbers for each relationship and put these negative numbers at the appropriate place in the list.

This excerpt from the grammar for the PLOT procedure illustrates the use of the @MARK semantic action to indicate the end of the PLOT request. Note that the REQUEST production is defined later in the grammar and also uses the @MARK semantic action to separate parts of the plot request.

```
PLOTSTMT    =                            @STMTINIT(4)
            ("PLOT"|"PLOTS$") REQUEST+
                                         @STMTLIST(1, 3)
                                         @MARK(5)
            < "/" PLOTPARM*>             @STMTEND,
```

@OPT

The @OPT semantic action defines an on/off option. Code the action as follows:

@OPT(*number*)

where *number* is from 0 to 63 and defines which of the 64 bits in the `opt` field of the STMT structure stores this option. If the terminal that names this option (for example, the DOUBLE option of PROC PRINT) is specified when the procedure is invoked, the corresponding bit is set to '1'. All bits are off ('0') by default.

@PARM

The @PARM semantic action defines a parameter, which is invoked with a *keyword= value* format. Code the @PARM action as follows:

@PARM(*field*<,*mode*,<*length*>>)

where

field is the index to the `fld` array.

mode indicates the type of data that the user can assign to the parameter. By default, the mode is 1. The value of the parameter is stored in different locations depending upon the value of the mode argument. Valid values are

1 permits a numeric value. This type of value is stored in the `fval` field of the STLIST.

2 permits an 8-byte character string. This value can be a valid SAS name, but that is not required. If the item is a quoted string, quotes are removed before the item is stored in the field. This type of value is stored in the `nval` field of the STLIST.

3 or 5 permits a character string of more than eight characters. By default, the length of the string is set to the length of the string specified by the user. (See *length*.) If the item is a quoted string, quotes are removed before the item is stored in the field. This type of value is stored in the `cval` field of the STLIST.

length is the length of mode 3 or mode 5 variables. If you do not specify this argument, the length is set to the length of the user's input string. If you specify *length*, the number of bytes of the user's input that is moved to `cval` is the smaller of *length* or the actual length of the

(length continued)

input. The n field of the FIELDSTR contains the actual number of bytes moved.

The `fld` element for a parameter is completed as follows: `type` is set to 2, and `mode` is equal to the second argument for the @PARM action.

@PARMVAL

The @PARMVAL semantic action stores an integer constant in the `fval` field of the STLIST. Code the action as follows:

```
@PARMVAL(field,value)
```

where

field is the index to the `fld` array.

value is the integer to be stored in the parameter. The integer value is written to the parameter field as a double.

The `fld` element for the parameter is completed as follows: `type` is set to 2, and `mode` is set to 1.

@PROCINIT

The @PROCINIT semantic action initializes various fields in the `proc` structure. The @PROCINIT action must be the first semantic action in the PROC statement in your procedure's grammar so it is performed as soon as the PROC statement is parsed. To create the statement structure for the procedure, you should code a @STMTINIT immediately after the @PROCINIT action. @PROCINIT has no arguments.

@RANGE

The @RANGE semantic action specifies a valid range for an input parameter. Code the action as follows:

```
@RANGE(<low,>high)
```

The *low* and *high* numbers must both be integers, and *low* must be less than or equal to *high*. The *low* and *high* numbers are included in the range of valid values. If only one number is specified, the value specified by the user must be less than or equal to *high*.

If the user specifies a value for the parameter that is out of the range, the error flag for semantic action errors is set, and the system issues an error message.

@SASNAM

The @SASNAM semantic action ensures that the current token is a valid SAS data set name with *number* levels in the name. SAS data sets can have up to three levels in the name in the format: *libref.member.type*. For example, a SAS data set can be named MYLIB.MYINPUT.DATA. Code this action as follows:

```
@SASNAM(number<,1>)
```

where

number is the number of levels permitted in the data set name. This number can be treated as the maximum number of levels or as the only number of levels permitted, depending upon whether you specifiy the second argument or not.

1 indicates how to interpret the first argument. If you specify this argument, *number* indicates the exact number of levels that must be specified in a data set name. If you omit this argument, *number* indicates the maximum number of levels that the user can specify.

@STMTEND

The @STMTEND semantic action finishes the statement and chains its statement structure to the chain of statement structures. Statements are chained in the order that they are invoked by the user. The name of the statement is stored in the `name` field of the statement structure. You can use @STMTEND only when the statement is started with @STMTINIT and only after all of the other semantic actions describing the statement. Refer to the description of the STMT (statement structure) in Chapter 20 for more information on chaining statement structures.

@STMTINIT

The @STMTINIT semantic action allocates and initializes a statement structure. Each grammar must include a @STMTINIT action for the PROC statement to initialize the structure in which the parser stores options, parameters, data set names, and variable lists. In addition, you can specify a @STMTINIT semantic action for any statement that needs a separate statement structure. Always end statements that begin with @STMTINIT with a @STMTEND. Code the action as follows:

```
@STMTINIT(nfields)
```

where *nfields* is the number of fields to allocate for the `fld` array in the statement structure. Be sure that the *nfields* value given is at least as large as the largest number assigned to any of the following types of semantic actions:

- @PARM (parameters)
- @STMTLIST or @LIST (lists)
- @DS or @DSDFLT (data sets)

- @LABEL (labels)
- @LISTCT, if the count is stored in a list not defined by the @LIST action
- @LISTVAL, if the value is stored in a list not defined by the @LIST action
- @PARMVAL (parameter values).

Some procedures require more than one statement structure, so multiple @STMTINIT actions are necessary. This occurs when a statement has information that cannot be stored appropriately in the PROC statement structure. For example, the PLOT statement of PROC PLOT or TIMEPLOT can occur many times. Therefore, it is best that each PLOT statement have its own statement structure. It is advisable also to create a separate structure for statements other than the procedure statement that permit the user to specify options and parameters in the statement.

Note: When multiple statement structures are created, they are chained together by the pointers in the `next` field of the statement structure. Refer to the description of the STMT (statement structure) in Chapter 20 for more information on chaining statement structures.

@STMTLIST

The @STMTLIST semantic action establishes a list in the statement structure as the one currently being processed. This action should follow either a @STMTINIT action, which creates a new statement structure, or a @STMTPROC action, to place the list in the procedure's statement structure. To actually add items to the list, you must use one of the productions, such as VARLIST, NUMLIST, ONEVAR, or others that define the contents of a list. Code this action as follows:

```
@STMTLIST(field,mode)
```

where

field is the index to the `fld` array in the statement structure.

mode indicates the type of data to be stored in the list. Note that the value you specify for *mode* in this action is not the same as the `mode` field in the STLIST. Table 20.1 in Chapter 20 summarizes the types of data and the values of the `mode` field stored in the statement structure when this action is processed. Valid values for this parameter are the following:

1 indicates numeric variables from a SAS data set. The user must specify valid variable names of numeric variables from the current SAS data set.

2 indicates character variables from a SAS data set. The user must specify valid variable names of character variables from the current SAS data set.

3 indicates mixed variables from a SAS data set. The user must specify valid variable names of any variables in the current SAS data set.

4 indicates numeric list. This list can contain integer or real values.

7 indicates name list. This list consists of valid SAS names.

The @STMTLIST action simply defines where to store the list and what types of values are stored. The actual work of storing the list values is performed by one of several standardized productions in the stub grammar. For example, variable lists are processed by the VARLIST production; other productions include ONEVAR, NUMLIST, and NAMELIST. The coding for a VAR statement would be similar to the following:

```
VARSTMT = @STMTPROC @STMTLIST(6,1)
          "VAR"  VARLIST ,
```

The @STMTLIST action defines the list, which consists of only numeric variables, as the sixth element of the `fld` array in the procedure's statement structure, and the VARLIST standard production performs the actions to store the variables in this list.

Note: The values stored in the first three types of lists are not the actual variable names entered by the user. The value that is actually stored for the variable name is a integer value that represents the relative position of the variable in the data set variable list.

@STMTPROC

The @STMTPROC semantic action directs the processor to store the information derived from parsing this statement in the procedure's statement structure. Use this action instead of @STMTINIT when you do not need to create a separate statement structure for the statement; this typically is the case when the statement simply lists variables, such as the VAR or CLASS statements. Do not use the @STMTEND action with the @STMTPROC action.

Special Semantic Actions

In some cases your grammar needs to alter the normal actions of the parser. Normally, the parser processes the statements entered by the user one word at a time. Each token can be interpreted individually without knowing what the following tokens are. There are some circumstances, however, where the context of a token must be known before the correct action can be taken. For example, in most cases parameters are entered by the user in the following format:

keyword=value

Your procedure, however, may allow a parameter to have a list of values instead of a single value, as in this example from the OUTPUT statement of the MEANS procedure:

```
OUTPUT MEAN=MA MB   STD=SA;
```

In this case, the parser needs to know when the first list of values ends and the next option or parameter begins so that the second parameter keyword is not treated as part of the first list of values. The @SPECIAL action indicates to the parser that it must wait to perform the appropriate actions until the context of the token is clear.

Note: Because you are altering the normal processing of the parser, you must indicate that you want to make a change *before any processing begins* for that token. This means that the @SPECIAL must come at least two tokens ahead of the word that requires special processing in the grammar definition. Consider this incorrectly coded example:

```
"VALUE" "=" @SPECIAL(1) NAME    /* incorrect example */
```

The special semantic would have no effect on the name that is entered because the processing path has already been determined by the time the name is analyzed. This statement is correctly coded as follows to permit the special processing to apply to the name value:

```
"VALUE" @SPECIAL(1) "=" NAME    /* correct example */
```

After you use a special semantic action, you must turn off the effects of that action to restore normal grammar processing. See the description of the @SPECIAL(1*nn*) semantic action later in the chapter.

@SPECIAL(1)

The @SPECIAL(1) semantic action is for grammatical forms like *keyword=variables* so that a word followed by an equal sign is taken to be a keyword for a parameter instead of a value in a list. For example, the grammar for PROC MEANS parses the following statement by using a NAMELIST to handle the XM and YM values:

```
OUTPUT MEAN=XM YM VAR=XV YV;
```

However, the parser does not know that VAR is a terminal for the next parameter. To the parser, VAR looks like another element in the list after XM and YM. To circumvent this problem, the @SPECIAL(1) semantic action is used in the grammar:

```
OUTPUTSTMT =                      @STMTINIT(20)
                                  @SPECIAL(1)
                 "OUTPUT"
                 OUTOPTS*         @SPECIAL(101)
                                  @STMTEND ,

OUTOPTS      = "N"      "="       @STMTLIST(1,6) NAMELIST
             | "MEAN"   "="       @STMTLIST(2,6) NAMELIST
             | "VAR"    "="       @STMTLIST(3,6) NAMELIST ,
```

The @SPECIAL(1) action is coded two terms before the parameters (OUTOPTS) to which it applies. Each of the parameters in this statement permits a list of values instead of a single value. The @SPECIAL(1) action notifies the parser to check if the token it is handling is followed by an equal sign. If not, the token is treated as one of the elements of the list; if the equal sign is present, the parser recognizes the token as the next terminal.

Note: The @SPECIAL(101) action turns off the @SPECIAL(1) action so that other parts of the grammar are not affected.

@SPECIAL(2)

This special action is like @SPECIAL(1) except that it applies to the left parenthesis, as well as the equal sign. Use this action when you want to use parentheses for purposes other than array references, which is the normal processing for parentheses. @SPECIAL(2) permits the parser to handle a statement like the following example from the SUMMARY procedure:

```
OUTPUT  MEAN=PREMEAN POSTMEAN  STD(POST)=STDPOST;
```

In this statement the terminals, such as MEAN and STD, can be followed by an opening parenthesis, (, as well as an equal sign, = .

@SPECIAL(14)

This special action causes the tokenizer to treat a minus or plus sign followed by a number as a signed number. The sign is concatenated to the number. This is the default processing for minus and plus signs. You must turn off this special action by using @SPECIAL(114) if you want your grammar to handle expressions or ranges of values such as the following:

```
MIN  x-4;

RANGE  10-100;
```

Then, at the end of the statement that permits the expressions, use @SPECIAL(14) to restore normal grammar processing.

@SPECIAL(1*nn*)

To turn off the effects of a special action that is needed in only a limited portion of your grammar, include the @SPECIAL action with an argument of 100 plus the number of the action to be turned off. For example, to turn off @SPECIAL(1), include this action at the end of the statement where the special action is used:

```
@SPECIAL(101)
```

Chapter 22 SAS_X Routines

Introduction

The SAS_X routines described in this chapter are a fundamental part of writing SAS procedures. These routines provide many facilities for interfacing with SAS data sets and the SAS Supervisor. Some of the SAS_X routines replace standard language commands for common tasks such as reading from and writing to data sets or for allocating memory. Other routines are specifically for writing SAS procedures, such as the routines that parse the statements entered by the users or perform BY-group processing.

The SAS_X routines are listed here alphabetically by groups. SAS_X routines are grouped by name so that those with a common prefix perform related actions.

Note: All SAS_X routines appear in this chapter in alphabetical order except SAS_XOREOPN, SAS_XORID2N, and SAS_XOUPD, which are in the SAS_XO group and therefore appear before the SAS_XOPT routines. Additionally, the routines work with all supported languages except where noted.

The description of each routine in this chapter provides the declarations for all the arguments needed by the routine. In most cases, it is advisable to declare the arguments as described here. When you include the portable declarations file (for example, in C the `uwproc.h` file) described in the appendix appropriate to the language you are using, you can use the variables exactly as illustrated.

Conventions for Declaring Variables

The examples in this chapter, as well as other examples throughout this guide, use data types that are common to the C language. In addition, other types have been defined within SAS/TOOLKIT files.

Refer to Table 10.1 in Chapter 10, "Writing a SAS Procedure," for the equivalent data types in FORTRAN, PL/I, and IBM 370 assembler languages.

Conventions for Using Parameters in SAS_X Routines

Each SAS_X routine described in this chapter has a section labeled "Declarations", which contains a table of variables you need to declare to call the SAS_X routine. The format of these tables is as follows:

- The Type column contains a C data type for each variable used by the routine. Note that the data types can be any of the native C types or the defined types listed in Table 22.1. Refer to Table 22.1 for the comparable data type in other languages.

- The Variable column lists each variable used by the routine. Note that in many cases a variable is declared as a pointer to another data type. You do not have to create a pointer variable and store the address of the other data. Instead, you can simply declare the data and use the address operator, the ampersand (&), in the call.* For example, a table may list a variable in the following format:

  `long` * *position-ptr*

  Your program can either declare a `long` and a pointer to a `long` and then assign the address of the `long` field to the pointer, or it can declare a `long` and use the address operator in the function call. These two techniques are shown here:

```
   /* declare both the data and the pointer */
ptr    file_ptr;
long   pos,
       * pos_ptr;

pos_ptr = &pos;
SAS_XCNOTE(file_ptr, pos_ptr);

   /* declare only the data and obtain the address of the data */
ptr    file_ptr;
long   pos;

SAS_XCNOTE(file_ptr, &pos);
```

* Keep in mind that you cannot use the address operator with arithmetic expressions or with the result of a function. That is, neither of the following expressions is valid: &(x/2) or &(log(10)).

In general, this book uses the first form to illustrate function calls in the "Usage" sections. Examples may use either form. Either form is acceptable.

- The Use column of the table shows how the variable is used by the SAS_X routine. The following values can appear in this column of the table:

  input — means that your procedure must supply a value for this parameter.

  output — means that the SAS_X routine passes a value back to your procedure. Note that because the C language does not permit a function to actually change the value of a parameter, you actually pass the address of a variable. The contents of the variable are changed, but the address remains the same. Therefore, the output is actually stored at the address of the variable. Keep in mind that the pointer listed in the table is not the actual output; the value pointed to is the output value.

  returned — is the function return value (or the *l-value* in C).

  update — is somewhat similar to the *output* values. In this case, your procedure sets a value and then passes the address of the value to the SAS_X routine. The SAS_X routine changes the value at the address it receives, but the address remains the same. The primary difference between output and update values is that update values have meaning before they are passed to the SAS_X or SAS_Z routine. An output value is useful only after the SAS_X or SAS_Z routine has set its value. Once again, keep in mind that the pointer listed in the table is not a value that gets updated; the value pointed to is what is updated.

- The Description column briefly describes the variable. Look for more details on how to use the variable in the "Description" section that follows the table.

Conventions for Representing Usage

In the sections that follow, each routine is illustrated in a section called "Usage". If the routine is a function, the conventions for calling the routine are the same for the C, PL/I, and FORTRAN languages. Note that for FORTRAN functions, you must omit the ending semicolon.

If the routine is a C function that returns a void, in the PL/I or FORTRAN language the routine is a CALL routine. In this case, the usage for all three languages is illustrated.

Conventions for Assembler Programming

You call SAS_X, SAS_Z, and other SAS/TOOLKIT routines from assembler in exactly the same way that the assembler code generated by the C compiler calls routines, which is using a call-by-value technique. All SAS/TOOLKIT routines described in this chapter and the next two chapters are referenced using macro variables. For example, the V-type address constant for the interface routine SAS_XFFILE is accessed via the macro variable &SAS_XFFILE.

Use standard register conventions when calling SAS/TOOLKIT routines:

R1 points to a parmlist. This parmlist must be aligned on a doubleword boundary, and any double constants written into the parmlist must be on doubleword boundaries. You must skip to a doubleword boundary if necessary.

R15 points to the V-type address constant of the calling routine.

R14 points to the return address.

R13 points to a save area with a minimum size of DSAMIN.

All registers are restored upon return except R15, which contains the return code from the SAS/TOOLKIT routine.

The following example illustrates a call to SAS_XFFILE in both C and IBM 370 assembler language.

C version

```
rc = SAS_XFFILE(fileid, 'F',
                NULL, &code);
```

Assembler version

```
* (R13 already points to DSA)
CALLIT   L      R14,fileid
         LA     R15,C'F'
         SLR    R0,R0
         LA     R1,code
         STM    R14,R1,parmlist
         LA     R1,parmlist
         L      R15,&SAS_XFFILE
         BALR   R14,R15
         ST     r15,rc
         .....
DSA      DS     0F,(DSAMIN)C
```

Assume that `fileid` is at address 045C0368 and `code` is at address 045C0400. When SAS_XFFILE is called, the `parmlist` that R1 points to contains the following:

```
045C0368 000000C6 00000000 045C0400
```

Unlike call-by-address languages, the VL bit does not need to be set. In fact, setting the VL bit is harmful if the last argument is not an address. Notice in the previous assembler example that the second argument, 'F', is passed as a 4-byte integer. This is because in the C language a single-byte character value enclosed in single quotes is promoted to the type `int`. An `int` is a fullword integer under the MVS and CMS operating systems, so the character 'F' (0xC6) becomes the fullword 000000C6. In this case, R0 contains the hex value 000000C6 to be stored in the parmlist.

Note: You cannot have a parmlist that is already created at assembly time unless it contains only addresses or constants as in the following example:

C version

```
rc = SAS_ZSTRNDX(string, 10,
                 string2, 2);
```

Assembler version

```
CALLIT   L     R1,parmlist
         L     R15,&SAS_ZSTRNDX
         BALR  R14,R15
         ST    R15,rc
         ....
         DS    0D
PARMLIST DC    A(string)
         DC    F'10'
         DC    A(string2)
         DC    F'2'
```

If the parmlist contains a value that is set in your program, you must store the value into the parmlist before calling the routine, as illustrated here in these two sets of examples:

C version

```
   /* first example */
p = SAS_XMPOOLC(XM_ISA, XM_OSA, 0);

   /* second example */
p2 = SAS_XMALLOC(p, 1000L, 0);
```

Assembler version

```
* first example
         LA    R1,xmpoolc_parmlist
         L     R15,&SAS_XMPOOLC
         BALR  R14,R15
         ST    R15,p
* second example
         L     R1,p
         ST    R1,xmalloc_parmlist
         LA    R1,xmalloc_parmlist
         L     R15,&SAS_XMALLOC
         BALR  R14,R15
         ST    R15,p2
```

Testing Return Codes from SAS_X Routines

It is good programming practice to test the return codes from the SAS_X routines. Unless otherwise indicated in the description of the routine, a nonzero return code indicates an error has occurred. If the error occurred when you called a SAS_XC, SAS_XD, SAS_XO, or SAS_XV routine, you can obtain descriptive messages for the error by passing the return code to the SAS_XPRLOG routine. Refer to the description of SAS_XPRLOG for more information.

The SAS_XM routines also provide special handling for errors that occur while you are attempting to allocate memory. For more information on these routines, refer to "SAS_XM Routines: Memory Management" later in this chapter.

SAS_XBY Routines: BY-Group Processing

If the grammar for your procedure defines a BY statement, you should read all observations by calling the BY-processing routines. There is no need to test whether or not the user specifies a BY statement when invoking the procedure. When you call the BY-processing routines and the user has not specified a BY statement, the entire data set is treated as a single BY group.

The following template illustrates the typical processing loop for handling BY groups and shows which SAS_XBY routines to call:

```
ptr fileid;

   /* position to new BY group */
while (SAS_XBYNEXT(fileid) != 0) {
   initialize, possibly print

      /* read observation */
   while (SAS_XBYGET(fileid) != NULL) {
      do processing
      }
   do computations or write output

      /* some procedures will also do the following */
      /* rewind to beginning of BY group */
   SAS_XBYRWND(fileid, 'B');

      /* read observation */
   while (SAS_XBYGET(fileid) != NULL) {
      do processing
      }
   do computations or write output
   }
```

Note that BY groups are handled at the head of the BY loop rather than at the end. BY routines permit you to handle BY groups in a single input data set or to synchronize the processing of multiple data sets.

SAS_XBYGET

Read an observation of a BY group

Declarations

Type	Variable	Use	Description
ptr	*file-pointer*	input	fileid of an open input file
ptr	*obs-ptr*	returned	pointer to observation buffer: NULL if you have reached the end of the BY group or the file

Usage

```
obs-ptr = SAS_XBYGET(file-pointer);
```

Description

After calling SAS_XBYNEXT to position to the beginning of a BY group, use SAS_XBYGET to read an observation in a BY group. It is important that you use SAS_XBYGET, not SAS_XOGETN, to read BY-group observations because the SAS_XO routines read observations without regard to BY processing.

The SAS_XBYGET routine returns a pointer to the buffer of the observation just read. At the end of the BY group, it returns a NULL.

Example

In many cases, you use the SAS_XBYGET routine to access the next observation in a BY group, but you call the SAS_XVGET routines to actually read the observation into the variables in your program. In this case, the SAS_XBYGET routine simply checks for the end of the BY group. The positioning for the calls to SAS_XVGET routines is illustrated in the following example:

```
ptr fileid;

   /* position to new BY group */
while (SAS_XBYNEXT(fileid) != 0) {

      /* read observation */
   while (SAS_XBYGET(fileid) != NULL) {
      do processing - may include calls to SAS_XVGET routines
   do computations or write output
   }
}
```

SAS_XBYLIST

Build BY list for secondary data set

Declarations

Type	Variable	Use	Description
ptr	*file-pointer1*	input	fileid of the primary input file
ptr	*file-pointer2*	input	fileid of the secondary input file
int	*rc*	returned	return code: 0 = all variables matched 1 = some variables in the first file either are not in the second file or have different attributes 2 = no variables matched

Usage

```
rc = SAS_XBYLIST(file-pointer1, file-pointer2);
```

Description

The SAS_XBYLIST routine allows you to test to see if both input data sets have common BY-variable names and attributes. If the value returned by calling SAS_XBYLIST indicates a match, you can then use SAS_XBYSYNC to process the files synchronously. XBYLIST is called any time after SAS_XVLIST is called for the primary data set and usually before beginning the SAS_XBYNEXT loop for the primary data set. SAS_XBYLIST must be called before calling any other BY-processing routines for the secondary data set. The SAS_XBYLIST routine should be called only once.

The SAS_XBYLIST routine actually builds a BY list for a secondary data set. The BY statement, as processed by the grammar, is only applied to the primary data set (the data set that has 1 for the third argument in the @DS semantic action). You must call SAS_XBYLIST for all other data sets on which BY processing needs to be performed.

Handling nonzero return codes

When you call SAS_XBYLIST, the return code indicates whether the list of variables and variable types match. If the value returned by SAS_XBYLIST is 1 or 2, you need to decide if it is possible to continue BY processing for the files. For example, consider the following data sets:

```
Primary data set

  var#  name      type      length     format      label
  ----  ----      ----      ------     ------      -----
     1  A         num       8          <none>      <none>
     2  B         char      1          <none>      A LABEL
     3  C         num       8          DATE.       START DATE
     4  D         num       8          <none>      <none>
     5  E         num       8          <none>      <none>

Secondary data set

  var#  name      type      length     format      label
  ----  ----      ----      ------     ------      -----
     1  C         num       8          DATE.       START DATE
     2  A         num       8          <none>      <none>
     3  D         char      8          $HEX.       <none>
     4  B         char      1          <none>      A LABEL
```

Now consider the following BY statements and the result of building a list for the secondary data set:

BY A B
: results in a successful BY list with the variable numbers of 2, 4. The return code is 0.

BY D
: does not produce a BY list because the variable D is numeric in the first data set and character in the second data set. The return code is 2.

BY C D
: works partially since C matches, but D does not. In this case, the return code is 1.

BY E
: does not work because E is not present in the secondary data set at all. The return code is 2. You may choose to have your procedure accept this and deal with the secondary data set without BY processing.

Note that it is not necessary for the relative positions of the variables to be the same between data sets.

When variable lengths, formats, or labels differ

If the lengths of the variables are not the same, you may not be able to match properly between the observations. The length of the smaller variable is used in comparison. For example, if X is length 4 in one data set and length 8 in the other, only the first 4 bytes are compared.

If the labels or formats in the data sets are different for a variable, the labels and formats for the primary data set are used in creating the default BY line.

SAS_XBYLIST *continued*

Example

The following example illustrates how to use SAS_XBYLIST, SAS_XBYNEXT, and SAS_XBYSYNC to match two files and process the matching BY groups. This example requires that all BY variables in the transaction file must occur in the master file.

```
ptr  trans,    /* fileid of transaction data set */
     master,   /* fileid of master data set */
     in_ptr1,  /* pointer returned by XVGETI for master data set */
     in_ptr2;  /* pointer returned by XVGETI for transaction data set */

short nvar;


   /* test to ensure the user specified a BY statement */
if (SFTYPE(proc.head, 0) == 0) {
   SAS_XPSLOG("%1z This proc must have a BY statement.");
   SAS_XEXIT(XEXITBYERROR, 0);
}

   /* test BY variables to be sure they match */
if (nvar > 0)
   if (SAS_XBYLIST(trans, master) !=0 ) {
      SAS_XPSLOG("%1z By variables must exist in both input data");
      SAS_XPSLOG("         sets and must have the same attributes.");
      SAS_XEXIT(XEXITBYERROR,0);
    }

call SAS_XVGETI to set up scatter read process for both files

   /* do the scatter read and build the report */
while (SAS_XBYNEXT(trans) > 0) {
   while (SAS_XBYGET(trans) != NULL) {
      SAS_XVGET(in_ptr2, NULL);
         process transaction record
      if (SAS_XBYSYNC(trans,master) == 0) {
         if (SAS_XBYGET(master) != NULL) {
            SAS_XVGET(in_ptr1, NULL);
              process master record
         }
      }
      else {
          SAS_XPSLOG("%2z There was no entry in the master file for");
          SAS_XPSLOG("         one of the transactions.");
      }
   }
}
```

SAS_XBYNEXT

Position to beginning of a BY group

Declarations

Type	Variable	Use	Description
ptr	*file-pointer*	input	fileid of an open input file
int	*rc*	returned	variable number of the BY variable that changed: 0 = EOF

Usage

```
rc = SAS_XBYNEXT(file-pointer);
```

Description

The SAS_XBYNEXT routine positions to the beginning of the next BY group. The first time it is called, it performs necessary initialization to handle BY-group processing. It is important that you do not read any observations before calling SAS_XBYNEXT the first time. After calling SAS_XBYNEXT, you can read observations by calling the SAS_XBYGET routine. At the end of the file, SAS_XBYNEXT returns a 0.

XBYNEXT can be called even if you have not finished processing or even reading any observations from a BY group. Suppose that you have read through a BY group once, and you need to reread that group to do selective processing. You can use SAS_XBYRWND to rewind to the beginning of the BY group, find the specific observation you need, and then call SAS_XBYNEXT to skip the remainder of that BY group and begin at the next group.

Each time you issue a call to SAS_XBYNEXT, it positions you at the beginning of the next BY group. This is particularly useful if the procedure encounters a serious error in the data of a BY group that prevents continued processing. SAS_XBYNEXT allows processing to start again with another BY group.

Each time you call SAS_XBYNEXT and it positions to a new BY group, the value returned by SAS_XBYNEXT indicates which of the variable values changed to start the new BY group. For example, suppose you are processing this BY statement:

```
BY A B C;
```

Also suppose that the data set contains this data:

```
OBS  A  B  C  D
  1  1  1  1  1
  2  1  1  1  2
  3  1  1  1  1
  4  1  1  2  1
  5  1  2  1  1
```

SAS_XBYNEXT *continued*

When you call XBYNEXT and it positions to the second BY group (beginning with observation 4), the value returned by SAS_XBYNEXT is 3, the relative position number of the variable C in the list of BY variables. Positioning to the third BY group with SAS_XBYNEXT returns the value 2, to indicate the second variable, B, has changed.

Note: The number that SAS_XBYNEXT returns is simply the position number of the variable in the BY list; it is not the position of the variable in the SAS data set. The first time SAS_XBYNEXT is called, it returns 1 unless the data set is empty, in which case it returns a 0.

The SAS_XBYNEXT routine can be called any number of times after reaching the end of file (and before rewinding). It will always return a 0 in such cases.

The SAS_XBYNEXT routine automatically handles the DESCENDING and NOTSORTED options. You do not need to be concerned with the ordering of the observations. If your procedure is performing BY-group processing on a data set that is not in sorted order (and NOTSORTED was not specified), the system automatically terminates the procedure and prints appropriate messages indicating out-of-sequence observations.

If the user does not specify a BY statement when invoking the procedure, the SAS_XBYNEXT routine returns a 0 every time after the first time it is executed.

Example

The call to SAS_XBYNEXT should be placed at the top of a loop that reads data as illustrated in the following example:

```
ptr  fileid;

   /* position to new BY group */
while (SAS_XBYNEXT(fileid) != 0) {
   initialize, possibly print

      /* read observation */
   while (SAS_XBYGET(fileid) != NULL) {
      do processing
   }
}
```

In this example, the first time the SAS_XBYNEXT routine is used, the value returned is 1, which permits the program to execute the nested loop. The SAS_XBYGET routine returns a 1 at the end of each BY group and at the end of file. If no BY statement is used, the next time the SAS_XBYNEXT statement is processed is at the end of file.

SAS_XBYRWND

Rewind to beginning of BY group or data set

Declarations

Type	Variable	Use	Description
ptr	*file-pointer*	input	fileid of an open input file
int	*rewind-type*	input	type of rewind: 'F' = to first observation in data set 'B' = to first observation in BY group
int	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc = SAS_XBYRWND(file-pointer, rewind-type);
```

Description

The SAS_XBYRWND routine moves backward through the data set to the beginning of the BY group being processed or to the beginning of the data set, depending upon how you call the routine. Note that you should only rewind to the beginning of a BY group when using a SAS_XBYNEXT loop. Use SAS_XBYRWND to rewind to the beginning of a data set before you enter such a loop.

Example

Use this routine when you need to do some preliminary processing of the data set before you begin the major actions of your procedure. For example, if you want to create an array of all observations in the data set, you can pass quickly through the data set counting the observations, and then rewind the data set to the beginning. With this technique you can allocate the correct amount of space for an

SAS_XBYRWND *continued*

array before you begin actually processing the data set. The following example illustrates this technique:

```
int numofobs;
ptr infile;

   /* count the number of observations in the data set */
numofobs = 0;
while (SAS_XBYNEXT(infile) != 0) {
   while (SAS_XBYGET(infile) != NULL) {
      numofobs++;
   }
}

   /* reset the file pointer to the begining of the data set */
SAS_XBYRWND(infile, 'F');
```

SAS_XBYSYNC

Synchronize primary and secondary data sets

Declarations

Type	Variable	Use	Description
ptr	*file-pointer1*	input	fileid of the primary input file.
ptr	*file-pointer2*	input	fileid of the secondary input file.
int	*rc*	returned	match code: 0 = positioned to matched BY group 1 = either the end of *file-pointer2* was reached, or *file-pointer2* does not have one of the BY variables that is in *file-pointer1* 2 = NOTSORTED option is specified, and the BY variables do not match exactly. If you specify that the data sets are not sorted, their BY variables must match exactly in order to use SAS_XBYSYNC.

Usage

```
rc = SAS_XBYSYNC(file-pointer1, file-pointer2);
```

Description

The XBYSYNC routine is used to synchronize the beginning of BY groups in two files. If a call to SAS_XBYLIST indicates that the two files have matching BY variables, use SAS_XBYNEXT to obtain the beginning of a BY group for the first file and then call XBYSYNC to find the same BY group in the second file. SAS_XBYSYNC reads through the secondary data set to match up with the BY group of the primary data set.

Example

The following sample code illustrates a typical use of SAS_XBYSYNC. In this example, all observations in the primary data set are processed before any of the observations in the matching BY group of the secondary data set are processed.

```
ptr  fileidp1,   /* primary data set   */
     fileidp2;   /* secondary data set */

   /* process a BY group */
while (SAS_XBYNEXT(fileidp1) != 0) {

      /* read from primary data set */
   while (SAS_XBYGET(fileidp1) != NULL) {
      process observation from primary data set
   }

      /* look for BY group in secondary data set */
   if (SAS_XBYSYNC(fileidp1, fileidp2) == 0) {

         /* read from secondary data set */
      while (SAS_XBYGET(fileidp2) != NULL) {
         process observation from secondary data set
      }
   }
}
```

Now, consider the effect of the following BY statement on the data sets illustrated in Figure 22.1 as they are processed by this example.

```
by color location;
```

SAS_XBYSYNC *continued*

Figure 22.1 *Synchronizing BY Groups*

Primary data set

Color	Location	Qty
red	top	5
red	bottom	7
blue	top	8
green	top	4
green	top	6
black	top	1

Secondary data set

Color	Location	Code
red	bottom	D5
red	bottom	84
blue	top	Z0
blue	bottom	R4
green	top	5T

The following list explains the processing steps performed by the sample code:

1. SAS_XBYNEXT positions to the first observation in the primary data set.
2. SAS_XBYGET(fileidp1) reads the first observation in the primary data set.
3. SAS_XBYSYNC indicates there is no matching BY group in the secondary data set for the first BY group in the primary data set (that is, Color=red, Location=top does not exist in the secondary data set). SAS_XBYGET is not called.
4. SAS_XBYNEXT positions to the next observation in the primary data set (Color=red, Location=bottom).
5. SAS_XBYGET(fileidp1) reads the second observation in the primary data set.
6. SAS_XBYSYNC finds a match in the secondary data set. Observation 1 in the secondary data set has Color=red, Location=bottom.
7. SAS_XBYGET(fileidp2) reads observations 1 and 2 from the secondary data set. These are all of the observations in the BY group.
8. SAS_XBYNEXT positions to the next observation in the primary data set (Color=blue, Location=top), and SAS_XBYGET(fileidp1) reads it.
9. SAS_XBYSYNC finds a match in the secondary data set and SAS_XBYGET(fileidp2) reads it.
10. SAS_XBYNEXT positions to the fourth observation in the primary data set (Color=green, Location=top).
11. SAS_XBYGET(fileidp1) reads observations 4 and 5 from the primary data set.
12. SAS_XBYSYNC skips over observation 4 (Color=blue, Location=bottom) in the secondary data set and finds a matching BY group in observation 5 (Color=green, Location=top).
13. SAS_XBYGET(fileidp2) reads observation 5 from the secondary data set.

14. SAS_XBYNEXT positions to observation 6 in the primary data set (Color=black, Location=top) and SAS_XBYGET(fileidp1) reads it.
15. SAS_XBYSYNC finds the end of the secondary file and indicates no matching BY group exists. SAS_XBYGET is not called.
16. SAS_XBYNEXT finds the end of the primary file.

▶ ***Caution*** .... ***Your procedure should always process a BY group from each data set before advancing to the next BY group in either data set.***
That is, your application may need to match BY groups differently than the process shown in this example, but to perform BY-group processing, you need to keep matching the BY groups from the two data sets. ▲

SAS_XC Routines: Utility Files

The SAS_XC routines provide the ability to read and write utility files. Utility files are SAS files that are usually stored in the WORK data library and used for scratch space. For example, if your procedure builds a large matrix of data values that you will need later in your program but you do not want to keep it in memory until you reach that part of the program, you can store the matrix in a utility file until you need it again. Utility files are temporary and should not be expected to be kept after the procedure terminates.

The advantages of using SAS_XC routines instead of native I/O routines (such as `getc/putc` in C, READ/WRITE in PL/I or FORTRAN) are as follows:

- The SAS_XC routines are portable across operating systems.
- The file cleanup is handled for you (that is, nothing remains after the SAS job terminates).
- The method is consistent across languages.

Note: Utility files are byte-oriented files. That is, there are no internal markers in utility files to indicate the length or end of records. The SAS_XC routines enable you to read, write, and point to records in a file, but you must maintain information about record lengths and locations manually. The descriptions of SAS_XCNOTE and SAS_XCWRITE illustrate methods for maintaining locations and records lengths.

SAS_XCCLOSE

Close a utility file opened by SAS_XCOPEN

Declarations

Type	Variable	Use	Description
ptr	*utility-file-ptr*	input	fileid returned from SAS_XCOPEN
int	*disposition*	input	file disposition: 0 = normal close XC_DELETE = delete after close XC_LEAVE = valid only for sequential files
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc = SAS_XCCLOSE(utility-file-ptr, disposition);
```

Description

This routine closes and, optionally, deletes a utility file. If you specify XC_DELETE for the *disp* parameter, the file is deleted after it is closed. Otherwise, the file is deleted by the SAS Supervisor after the procedure ends. .

Example

This example shows how to open a file for processing and then close it. Note that the disposition for closing the file indicates it should be deleted when closed.

```
struct XCOPSTR copn_str;
rctype  rc;
ptr     statfile;

   /* set up the utility file  */
memcpy(copn_str.fn.libname, "        ", 8);
memcpy(copn_str.fn.memname, "TMPDAT  ", 8);
memcpy(copn_str.fn.memtype, "UTILITY ", 8);
copn_str.max_rc = 0;
copn_str.mode = XC_UPDATE + XC_RAND;
```

```
   /* open the utility file and check return code */
if (rc = SAS_XCOPEN(&statfile, &copn_str)) {
   SAS_XPRLOG(rc);
   SAS_XPSLOG("%1z Could not open Utility file.
The Utility file must be opened for this PROC to run.");
   SAS_XEXIT(XEXITIO, 0);
}
process file
SAS_XCCLOSE(statfile, XC_DELETE);
```

SAS_XCNOTE

Return the next byte position within a utility file

Declarations

Type	Variable	Use	Description
ptr	*utility-file-ptr*	input	fileid returned from SAS_XCOPEN
long	**position-ptr*	output	the current byte position of the file
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc = SAS_XCNOTE(utility-file-ptr, position-ptr);
```

Description

SAS_XCNOTE routine notes the position of the next record in a file so you can return to the record. The value returned by SAS_XCNOTE is the zero-based offset of the next byte in the file. The next byte is defined as the location of the next SAS_XCREAD or SAS_XCWRITE.

Note: If you may need to return to a record in the file, call SAS_XCNOTE before you call SAS_XCREAD so you have the location if it is needed.

Calling SAS_XCNOTE on a file that has just been opened returns 0.

SAS_XCNOTE *continued*

Example

The following example writes a utility file, repositions to the beginning of the file, reads a record and then rewrites the record:

```
ptr   infile,
      statfile,
      ptr_xv;
long  pos;
short nvar;
int   x;
struct OBSSTR {
   define observation
} observat, tmpobservat;

   /* build the utility file within the BY processing */
while (SAS_XBYNEXT(infile) > 0) {
   while (SAS_XBYGET(infile) != NULL) {
      SAS_XVGET(ptr_xv, NULL);
      SAS_XCWRITE(statfile, (ptr)&observat, sizeof(observat));
   }
}

   /* reset the file pointer to the beginning of the file */
SAS_XCPOINT(statfile, XC_BEGIN);

for (x = 0; x < nvar; x++) {
      /* save the current position in the utility file */
   SAS_XCNOTE(statfile, &pos);
   SAS_XCREAD(statfile, (ptr)&tmpobservat, sizeof(tmpobservat));
   process utility file record

      /* write data back to Utility */
   SAS_XCPOINT(statfile, pos);
   SAS_XCWRITE(statfile, (ptr)&tmpobservat, sizeof(tmpobservat));
}
```

SAS_XCOPEN

Open a byte-oriented utility file

Declarations

Type	Variable	Use	Description
ptr	**ptr-to-util-file-ptr*	output	pointer to the fileid for the file opened by SAS_XCOPEN.
struct XCOPSTR	**ptr-to-xcopnstr*	input	open parameters. Refer to the "Description" section for information.
rctype	*rc*	returned	return code: 0 = success otherwise = unsuccessful.

Usage

```
rc = SAS_XCOPEN(ptr-to-util-file-ptr, ptr-to-xcopnstr);
```

Description

You open a utility file for input or output with the SAS_XCOPEN routine. If the file is opened successfully, you can read from it using SAS_XCREAD or write to it using SAS_XCWRITE. At any point, you can note your location in the file with SAS_XCNOTE.

You pass the parameters required for opening the file by completing fields in the XCOPSTR structure. For C procedures, this data structure is defined in the `uwproc.h` header file. For PL/I procedures, you must declare the XCOPSTR, which is used by SAS_XCOPEN. To do this, include the following statements in your program:

```
%INCLUDE XCOPSTR;
DCL   XCOPSTRPTR  PTR;
XCOPSTRPTR = ADDR(XCOPSTR);
```

You must complete the following fields in the XCOPSTR data structure:

- `fn`, which provides the complete libref, member name, and member type
- `mode`, which gives instructions for opening the file
- `max_rc`, which indicates whether or not the procedure should automatically end if an I/O error occurs.

Refer to the description of XCOPSTR in Chapter 20, "Data Structures," for more information on the structure.

SAS_XCOPEN *continued*

Example

The following example illustrates opening a utility file. Note how the XCOPNSTR is built.

```
ptr     statfile;
struct XCOPSTR copn_str;

   /* set up the utility file  */
memcpy(copn_str.fn.libname, "        ", 8);
memcpy(copn_str.fn.memname, "TMPDAT  ", 8);
memcpy(copn_str.fn.memtype, "UTILITY ", 8);
copn_str.max_rc = 0;
copn_str.mode = XC_UPDATE | XC_RAND;

   /* open the utility file and check return code */
if (rc = SAS_XCOPEN(&statfile, &copn_str)) {
   SAS_XPRLOG(rc);
   SAS_XPSLOG("%1z Could not open Utility file.
The Utility file must be opened for this PROC to run.");
   SAS_XEXIT(XEXITIO, 0);
}
```

SAS_XCPOINT

Set byte position within a utility file

Declarations

Type	Variable	Use	Description
ptr	*utility-file-ptr*	input	fileid returned from SAS_XCOPEN
long	*position*	input	byte position within the file: XC_BEGIN = beginning of file XC_END = end of file XC_TRUNC = truncate the file
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc = SAS_XCPOINT(utility-file-ptr, position);
```

Description

The SAS_XCPOINT routine points to a specific record in a file so you can read or write to the record. In most cases, you use the value returned by a previous call to SAS_XCNOTE as the *position*. The call to SAS_XCPOINT sets the location for the next SAS_XCREAD or SAS_XCWRITE. SAS_XCPOINT does not perform any I/O.

On some systems, the note value may not be an exact zero-based offset. Therefore, do not specify a constant for the *position* parameter of SAS_XCPOINT; always use *position* values that are returned by SAS_XCNOTE.

Example

Suppose that you call SAS_XCNOTE before reading from a file, store the value returned by SAS_XCNOTE (which is 0), and then read 5 bytes of the file with SAS_XCREAD. You then call SAS_XCNOTE again to obtain a note value. It returns the location of the second record because you are now positioned at the end of the record you just read. It is important to remember that if you want to position at the beginning of a record, you need to call SAS_XCNOTE *before* reading the record. Note the difference in the following example:

```
ptr  fileid;
long note1,
     note2;

   /* this sequence stores the location of the second record,  */
   /* the record after the first one that is read             */
SAS_XCREAD(...);
SAS_XCNOTE(fileid, &note2);
SAS_XCPOINT(fileid, note2);

   /* this sequence stores the location of the first record,   */
   /* the record that is read by this code                     */
SAS_XCNOTE(fileid, &note1);
SAS_XCREAD(...);
SAS_XCPOINT(fileid, note1);
```

Also see the example for SAS_XCNOTE for another example of using SAS_XCPOINT.

SAS_XCREAD

Read from a byte-oriented utility file

Declarations

Type	Variable	Use	Description
ptr	*utility-file-ptr*	input	fileid returned from SAS_XCOPEN
ptr	*buffer-ptr*	input	buffer to read into
long	*number-bytes*	input	the number of bytes to read
rctype	*rc*	returned	return code: 0 = successful nonzero = end of file

Usage

```
rc = SAS_XCREAD(utility-file-ptr, buffer-ptr, number-bytes);
```

Description

The SAS_XCREAD routine reads a record from a utility file and stores it in an area defined by your procedure. SAS_XCREAD reads from the current location in the file, which is maintained internally. That is, each time you read from the file, the current location is updated to point to the next byte to be read. You can change the current location by using the SAS_XCPOINT routine. You can also keep track of the current location of a record by using SAS_XCNOTE. Keep in mind that SAS_XCNOTE records the location of the next byte to be read, so if you want to return to a record later, you should call SAS_XCNOTE *before* you read the record.

Example

This example illustrates how to read a record from a utility file with fixed-length records. This example writes the utility file, rewinds to the beginning of the file and then reads and processes the file sequentially. For an example of how to read a utility file with varying-length records, refer to the example in the description of SAS_XCWRITE.

```
ptr    infile,
       ptr_xv,
       statfile;
double observat[],
       tmpobservat[];
short  nvar;
int    x;

use SAS_XMEM to allocate observat and tmpobservat buffers

   /* build the utility file within the BY processing */
while (SAS_XBYNEXT(infile) > 0) {
   while (SAS_XBYGET(infile) != NULL) {
      SAS_XVGET(ptr_xv, NULL);
      SAS_XCWRITE(statfile, &observat, sizeof(observat));
   }
}

   /* rewind file to the beginning */
SAS_XCPOINT(statfile, XC_BEGIN);

   /* read utility file and process */
for (x = 0; x < nvar; x++) {
   SAS_XCREAD(statfile, &tmpobservat, sizeof(tmpobservat));
   do processing
}
```

SAS_XCWRITE

Write to a byte-oriented utility file

Declarations

Type	Variable	Use	Description
ptr	*utility-file-ptr*	input	fileid returned by SAS_XCOPEN
ptr	*buffer-ptr*	input	buffer to write from
long	*number-bytes*	input	the number of bytes to write
rctype	*rc*	returned	return code: 0 = successful nonzero = I/O error (usually disk full)

Usage

```
rc = SAS_XCWRITE(utility-file-ptr, buffer-ptr,number-bytes);
```

Description

The SAS_XCWRITE routine writes a record to the file. SAS_XCWRITE writes to the current location in the file, which is maintained internally. That is, each time you write to the file, the current location is updated to point to the next byte to be written. You can change the current location by using the SAS_XCPOINT routine. You can also keep track of the current location of a record by using SAS_XCNOTE. Keep in mind that SAS_XCNOTE records the location of the next byte to be written, so if you want to return to a record later, you should call SAS_XCNOTE *before* you write the record.

If you are creating a utility file that has records of varying lengths, it will be easier to read the contents of the file if you store the length of each record before you write the record. For example, the following illustrates a file containing multiple records of varying lengths:

```
XXXXX        /* record one   */
YYYYYY       /* record two   */
ZZZ          /* record three */
```

You created this file by calling SAS_XCWRITE three times with lengths of 5, 6, and 3. But keep in mind that utility files are byte-oriented files, so the actual contents of the file look more like the following illustration:

```
XXXXXYYYYYYZZZ
```

There are no end-of-record markers or lengths for each record. To read the records that you write, you need to know the size of the logical record. If all records are of the same size, this is no problem. You simply specify a constant value for the *number-bytes* argument.

If you are writing varying-length records, before you write each record of data, write a fixed-length record that stores the length of the data record. In most cases, you can store the record length in a long integer. Then follow the length record with the stream of bytes for the data record.

This illustration uses `LLLL` to represent a long integer that contains the length of the data record that follows it.

```
LLLL        /* contains 5, length of XXXXX  */
XXXXX
LLLL        /* contains 6, length of YYYYYY */
YYYYYY
LLLL        /* contains 3, length of ZZZ    */
ZZZ
```

When you need to read the record with SAS_XCREAD, you can read the size of the record first and then know how many bytes of data to read.

The sample code for writing and then reading this type of utility file is provided in the following example.

Example

This example illustrates the method of writing and then reading utility files with varying-length records:

```
long length;
char record[80];
ptr  fileid;
int  x;

   /* writing the file */
length = 5;
SAS_XCWRITE(fileid, &length, sizeof(length));
SAS_XCWRITE(fileid, "XXXXX", 5);
length = 6;
SAS_XCWRITE(fileid, &length, sizeof(length));
SAS_XCWRITE(fileid, "YYYYYY", 6);
length = 3;
SAS_XCWRITE(fileid, &length, sizeof(length));
SAS_XCWRITE(fileid, "ZZZ", 3);
SAS_XCPOINT(fileid, XC_BEGIN);

   /* reading the file */
for (x = 1; x <= 3; x++) {
   SAS_XCREAD(fileid, &length, sizeof(length));
   SAS_XCREAD(fileid, record, length);
   SAS_XPSLOG("...%*s...", length, record);
}
```

Refer to the example for SAS_XCREAD for a simple illustration of writing a utility file with fixed-length records.

SAS_XD Routine: Directory Level Access

SAS_XDNAME is the only routine included with SAS/TOOLKIT software for accessing directories of SAS data libraries.

SAS_XDNAME

Return a list of members in a library

Declarations

Type	Variable	Use	Description
char8	*libname*	input	library to be examined.
char8	*member-type*	input	types to be listed: "ALL" = all members in the SAS data library "DATA" = all SAS data files in the library "VIEW" = all SAS data views in the library
struct MEMLIST	***ptr-to-memlists*	update	input is NULL. SAS_XDNAME returns a pointer to an array of structures allocated and filled in by SAS_XDNAME().
int	**number-ptr*	output	number of members listed.
rctype	rc	returned	return code: 0 = file exists XHWSEQ = access to directory one member at a time X_WEOF = no more directory items to return for sequential library otherwise = unsuccessful.

Usage

```
rc = SAS_XDNAME(libname, member-type, ptr-to-memlists, number-ptr);
```

Description

The SAS_XDNAME routine provides access to the directory information of a SAS data library. Most procedures do not need to call this routine, because the procedure accepts as input a single data set. However, you may need to determine a list of all data sets in a data library. SAS_XDNAME can provide a complete list of members, including SAS data sets, catalogs, views, and other member types that can be stored in a SAS data library, but the capability of processing member types other than SAS data sets is beyond the scope of procedures written with SAS/TOOLKIT software.* In most cases, you simply request that SAS_XDNAME return only the list members of type DATA or VIEW in a library.

The first time that you call SAS_XDNAME, pass a null pointer for the value of the *memlist* parameter. SAS_XDNAME updates the value of the *memlist* parameter to point to an array of structures. As long as the library is not a sequential library, the structure pointed to by *memlist* describes all of the members in the library that match the *memtype* parameter. Each structure in the list has the following form:

```
struct MEMLIST {
       char8  mem;
       char8  ext;
       int    unused;
}
```

You need to check the *rc* code returned by SAS_XDNAME. If the value of this return code is XHWSEQ, the input library is a sequential library. When you call SAS_XDNAME and the input library is a sequential library, SAS_XDNAME updates the value of the *memlist* parameter to point to an array of one MEMLIST structure. On the first call, SAS_XDNAME returns information about the first member in the data library in this structure. To gather all of the information about a sequential library, you must continue to place calls to SAS_XDNAME until the return code is X_WEOF. On subsequent calls to SAS_XDNAME, use the address returned in *memlist* on the first call and the storage space will be reused.

If a sequential library contains no members, the XHWSEQ return code is returned but the integer pointed to by `nlist` is set to 0. That is, the XHWSEQ return code takes precedence over the X_WEOF return code.

* SAS_XDNAME does not list utility files that are stored in a SAS data library. Since these files are considered to be temporary working files, they are not reported to users by SAS_XDNAME (or the DIR window of the SAS Display Manager System).

SAS_XDNAME *continued*

Example

This example illustrates how to obtain a list of all SAS data files in a SAS data library, open each one for input, obtain information about it, and search for particular items.* In this example, the procedure searches a directory and looks for all SAS data files that have a variable called ID:

```
struct MEMLIST *memlist;
struct NAMESTR *np;
struct XOPNSTR xopnstr;
int nlist,
    i,
    j,
    nvars,
    found;
ptr fileid;
char8  libname = "SASD    ";
rctype rc;

   /* get the member list for non-sequential files */
if ((rc = SAS_XDNAME(libname, "DATA    ", &memlist, &nlist)) == 0) {
   memcpy(xopnstr.libname, libname, 8);
   xopnstr.opnmode = XO_INPUT + XO_RAND;
   xopnstr.maxrc = 0;

      /* read through each member item in list */
   for (i = 0; i < nlist; i++) {

         /* fill in XOPNSTR */
      memcpy(xopnstr.memname, memlist[i].mem, 8);

         /* open the file */
      SAS_XOOPEN(&fileid, &xopnstr);
      SAS_XOINFO(fileid, XO_NVARS, &nvars);

         /* gather information on variables */
      for (j = 1; j <= nvars; j++) {
         found = 0;
         SAS_XVNAME(fileid, j, &np);

            /* look for ID variable */
         if (memcmp(np->nname, "ID      ", 8) == 0)
            SAS_XPSLOG("nname = %*s", 8, np->nname);
         }
```

* The CONTENTS procedure uses this method to obtain the member list, open each member, and print information on each variable.

```
      SAS_XOCLOSE(fileid, 0);
      }
   }

      /* get the member names for sequential files */
   else {
      memcpy(xopnstr.libname, libname, 8);
      xopnstr.opnmode = XO_INPUT + XO_SEQ;
      xopnstr.maxrc = 0;
      for (; rc != X_WEOF;) {

            /* fill in XOPNSTR */
         memcpy(xopnstr.memname, memlist[i].mem, 8);

            /* open the file */
         SAS_XOOPEN(&fileid, &xopnstr);
         SAS_XOINFO(fileid, XO_NVARS, &nvars);

            /* gather information on variables */
         for (j = 1; j < nvars; j++) {
            found = 0;
            SAS_XVNAME(fileid, j, &np);

               /* look for ID variable */
            if (memcmp(np->nname, "ID      ", 8) == 0)
               SAS_XPSLOG("nname = %*s", 8, np->nname);
         }
         rc = SAS_XDNAME(libname, "DATA    ", &memlist, &nlist);
         SAS_XPSLOG("rc = %ld", rc);
         SAS_XOCLOSE(fileid, 0);
      }
   }
```

SAS_XE Routines: Exit and Errors

The SAS_XE routines allow you to issue error messages, count errors that occur when the errors are not serious, and exit from the procedure with a variety of return codes to indicate the relative success of the procedure.

SAS_XEBUG

Print out programmer-generated bug messages

Declarations

Type	Variable	Use	Description
char	**message-ptr*	input	pointer to a string containing the message

Usage

C	`SAS_XEBUG(message-ptr);`
FORTRAN	`CALL SAS_XEBUG(message-ptr)`
PL/I	`CALL SAS_XEBUG(message-ptr);`

Description

The SAS_XEBUG routine is for testing your procedure. Do not use SAS_XEBUG for errors entered by the user or for invalid data. This routine produces an error in the following format:

```
ERROR: A bug in SAS has been encountered. Please call your SAS
       representative and report the following message:
       message
       Save your printouts.
```

Please be sure that your SAS Software Representative is aware of the procedures you have written so you can provide the necessary support.

In most cases, after calling SAS_XEBUG, you call SAS_XEXIT to end the program.

Example

The following example tests the return code from SAS_XSPARSE to be sure it is 0. Note that a nonzero return code indicates something is wrong with the grammar, not that the user specified incorrect statements. You must test the `proc.error` field to verify that the user's input is correct.

```
ptr mygramc  U_PARMS((void));

UWPRCC(&proc);
if (SAS_XSPARSE(mygramc(), NULL, &proc))
   SAS_XEBUG("Error in grammar");

if (proc.error >= XEXITPROBLEM)
   SAS_XEXIT(XEXITSYNTAX, 0);
```

SAS_XEXIT

Exit from procedure; issue message if necessary

Declarations

Type	Variable	Use	Description
int	*exit-code*	input	the exit code.
int	*reason*	input	not currently used. Set this to 0.

Usage

C	`SAS_XEXIT(`*exit-code*`, `*reason*`);`
FORTRAN	`CALL SAS_XEXIT(`*exit-code*`, `*reason*`)`
PL/I	`CALL SAS_XEXIT(`*exit-code*`, `*reason*`);`

Description

A procedure exits by calling SAS_XEXIT. SAS_XEXIT is used for both normal exits and error exits. Procedures should not stop execution in any other way; that is, do not execute standard programming statements such as STOP, RETURN, or END to end a program in the language you are using. To exit from a procedure normally, use the following statement:

```
SAS_XEXIT(XEXITNORMAL, 0);
```

The error codes passed to SAS_XEXIT cause the SAS System to stop processing the remaining steps in the job that called your procedure. The remaining steps are treated as if the options OBS=0 and NOREPLACE are in effect.

The following symbolic names represent the valid return code values you can pass to SAS_XEXIT:

- XEXITABEND
- XEXITBUG
- XEXITDATA
- XEXITERROR

SAS_XEXIT *continued*

- XEXITIO
- XEXITMEMORY
- XEXITNORMAL
- XEXITPROBLEM
- XEXITSEMANTIC
- XEXITSYNTAX.

XEXITNORMAL has the value 0 and indicates a normal exit. All other return codes are explained in the following sections.

Severe Error Codes

These error codes cause the SAS System to stop processing the remaining steps in the job that called your procedure.

XEXITABEND
: causes the SAS job to abnormally terminate.

XEXITBUG
: indicates a bug in your program has been found. Before issuing the call to SAS_XEXIT with this return code, you should call SAS_XEBUG and indicate where the bug was detected.

XEXITDATA
: indicates the input data set contains invalid data that prevent further processing of the data set. Use this code when

  - a required variable does not exist in the input data set.
  - no variables in the data set meet the selection criteria. For example, the procedure processes only numeric variables, and the data set has only character variables.
  - the data set was opened with the wrong TYPE= option.

  Note: Do not use this return code when the call to SAS_XOINFO indicates that there are no observations or variables in the data set. This condition can occur when the user specifies OBS=0 to check the syntax of a job. The procedure should not issue an error message in this case. Use XEXITNORMAL in this case.

XEXITERROR
: indicates any other error not included in codes described in this list that prevents any further processing.

XEXITIO
: indicates that an I/O error has occurred that prevents further processing. Use this if your attempt to open an external file fails.

XEXITMEMORY
: indicates that the procedure has run out of memory and cannot recover or proceed. Before you call SAS_XEXIT, call SAS_XPSLOG and indicate how much memory was requested when the allocation failed (this amount is the value of the parameter passed to SAS_XMEMZER or SAS_XMEMGET when the allocation failed).

XEXITPROBLEM
: indicates an error condition has occurred. The best use for this code is as a test value, not for printing a message. See the example in the description of XEXITSYNTAX for an illustration of how to use XEXITPROBLEM to test for an error.

XEXITSEMANTIC
: indicates that an error in the meaning of a statement has occurred. For example, use this code when

- a data set is not found
- a variable is not found in the data set
- there are invalid combinations of options and parameters.

XEXITSYNTAX
: indicates a syntax error has been detected. In most cases, your procedure does not directly use this code. This code is used primarily by the parser. Your procedure accesses the return code from the parser to determine if any syntax errors occurred. The following is an example:

```
if (proc.error >= XEXITPROBLEM)
   SAS_XEXIT(XEXITSYNTAX, 0);
```

Syntax errors are errors in the format of a statement that prevent it from being parsed correctly. A misspelled option name is a syntax error.

Example

The first example illustrates how to exit from the procedure if a syntax error occurs during grammar processing.

```
ptr mygramc  U_PARMS((void));

SAS_XSPARSE(mygramc(), NULL, &proc);
if (proc.error >= XEXITPROBLEM)
   SAS_XEXIT(XEXITSYNTAX, 0);
```

This example illustrates a normal exit from the procedure.

```
SAS_XEXIT(XEXITNORMAL, 0);
```

SAS_XF Routines: Formats and Informats

Use the SAS_XF routines to access SAS formats and informats. The formats and informats can be any of those supplied by SAS Institute or created by the FORMAT procedure or SAS/TOOLKIT software.

When your procedure reads a SAS data set, it needs to check to see if the variables have formats associated with them. Use the SAS_XFFILE procedure before calling SAS_XVNAME to access the complete information on the formats and informats associated with the data set. You must call SAS_XFFILE before you begin calling the SAS_XVGET routines.

In addition, you can use SAS_XFNAME to add formats and informats to variables you are creating in an output data set. Refer to the description of the NAMESTR structure in Chapter 20 for information on how SAS_XVNAME, SAS_XFFILE, and SAS_XFNAME store information about formats and informats for a variable.

You may also want to use standard SAS formats or user-written formats to format data within the procedure. Note that these routines are not necessary when you simply want to use the formats associated with a data set. These routines are needed only when you want to format data created by your procedure. To call a format or informat, use one of the following routines:

SAS_XFXIC	calls a character informat.
SAS_XFXIN	calls a numeric informat.
SAS_XFXPC	calls a character format.
SAS_XFXPN	calls a numeric format.

Note: Before using these routines, you should decide whether or not your application warrants the extra effort that these routines take. For example, if you are going to read character strings that are always in the form of integers (such as the string '123'), you can use the SAS_ZSTOL routine to convert the string into a long integer. If you wish to perform a simple format of a `long` (or a double cast to a `long`), you can use the SAS_ZLTOS routine.

SAS_XFDEL

Delete loaded formats and informats

Declarations

Type	**Variable**	**Use**	**Description**
long	*iffcode*	input	format code to delete. This value is the `nfcode` from the NAMESTR. Use 999 to delete all loaded formats.

Usage

C	`SAS_XFDEL(iffcode);`
FORTRAN	`CALL SAS_XFDEL(iffcode)`
PL/I	`CALL SAS_XFDEL(iffcode);`

Description

SAS_XFDEL deletes formats or informats that you have loaded by calling SAS_XFNAME. It is not necessary to use SAS_XFDEL in your procedure because all loaded formats are freed when you call SAS_XEXIT. If you want to free a specific format, specify the format code. If you want to delete all loaded formats, specify 999 as the format code.

Example

Refer to the example for SAS_XFNAME for an illustration of SAS_XFDEL.

SAS_XFFILE

Obtain format codes for a list of variables

Declarations

Type	Variable	Use	Description
ptr	*file-ptr*	input	fileid of an open input data set.
int	*format-type*	input	type of formats to load. For PL/I and FORTRAN, declare this variable as a 1-byte character string. Valid values are 'F' = format 'I' = informat 'B' = both.
short	**list-ptr*	input	pointer to a list of variable numbers or NULL for all variables in the data set.
int	*number-in-list*	input	number of variable numbers in the list pointed to by *list-ptr* or 0 if *list-ptr* is NULL.
int	*rc*	returned	return code: 0 = successful nonzero = unsuccessful. This usually means that a format name associated with a variable does not exist. **Note:** SAS_XFFILE issues appropriate error messages in this case.

SAS_XFFILE *continued*

Usage

```
rc = SAS_XFFILE(file-ptr, format-type, list-ptr, number-in-list);
```

Description

The SAS_XFFILE routine should be called by most procedures that read SAS data sets and create output. The SAS_XFFILE routine ensures that when your procedure writes output, it formats the output appropriately. When you call SAS_XFFILE, it provides format codes that you need to be able to use SAS formats with data. You must call SAS_XFFILE before your procedure calls SAS_XVGET1V or SAS_XVGETD with the XV_FMT flag. (The XV_FMT flag requires that you have a valid format code.) If you try to use an informat or format that has not been loaded, the system issues an XEBUG message.

In addition to storing information about the formats associated with a variable, SAS_XFFILE also stores the code for the informat associated with the variable. The informat is useful only if you need to read data and store them using the same informat as an existing variable.

When you open a SAS data set either by using the @DS semantic action or by calling SAS_XOOPEN, the system creates a variable name structure (NAMESTR) for each variable in each variable list. Information about the variable, including the name of the format and informat associated with the variable, is stored in the NAMESTR. The following fields of the NAMESTR are completed when you open a SAS data set:

- `nname`
- `nform`*
- `nlabel`
- `niform`
- `ntype`
- `nlng`
- `nfl`
- `nfd`
- `nifl`**
- `nifd`.

* If a format is specified in a FORMAT or ATTRIB statement, the format name is stored in the `nform` field. The *w.d* format is an exception, however. No name is stored when this format is specified, but the length of the format is stored in `nfl`.

** The length of the informat is stored in `nifl`. If the informat is given and no width is specified, `nifl` is the default length of that informat. If no informat is given, `nifl` is 12 for numeric variables; for character variables the default value of `nifl` is equal to `nlng`.

Refer to the description of the NAMESTR in Chapter 20 for a complete description of these fields.

To determine if a format is associated with a variable, test the values in `nfl` and `nform`. If `nfl` is greater than 0 or `nform` is not blank, a format is associated with this variable. Similarly, to determine if an informat is associated with a variable, test the values in `nifl` and `niform`. If `nifl` is greater than 0 or `niform` is not blank, an informat is associated with this variable.

Although the NAMESTR contains some information about formats and informats for variables when the data set is opened, you must call SAS_XFFILE to obtain complete information about the format before you can use the format with the variable. A single call to SAS_XFFILE completes all of the formatting information about all variables in the list. You can call SAS_XFFILE multiple times if you have more than one list of variables that needs to be formatted.

The information added to the NAMESTR by the call to SAS_XFFILE depends upon how you call SAS_XFFILE:

- When you call SAS_XFFILE with a *format-type* of F or B, the length of the format is stored in `nflen`.* If a format with a specified width is associated with the variable, the value of `nflen` is the same as the value of `nfl`. If the format is given and no width is specified, `nflen` is the default length of that format. If no format is given, `nflen` is 12 for numeric variables; for character variables the default value of `nflen` is equal to `nlng`, the length of the variable. Special formats, such as $HEX, $BINARY, and $OCTAL, have different default widths. For example, if the format is $HEX and no length is specified, `nflen` is twice that of `nlng`.
- When you call SAS_XFFILE with a *format-type* of F or B, the correct access code is stored in `nfcode`. Access codes corresponding to default formats are used for those variables with no format name attached.
- When you call SAS_XFFILE with a *format-type* of I or B, the correct access code is stored in `nifcode`. Default informat codes are used for those variables with no informat name attached.

The format and informat codes are numeric codes that correspond to a format module logically loaded by the system. Each time a format is loaded by the system, it is assigned a format code. Format codes are not permanently associated with formats; instead they are assigned each time the system runs in the order that the formats are used.

When you use the format in a SAS_X routine call, you must specify the format code in the argument list. Since format codes are multiples of 100 and can become large very quickly, they are always typed as `long`.

Example

This example illustrates calling SAS_XFFILE to add formats to the NAMESTR information for all the variables in the list being processed. After the call to SAS_XFFILE, the `nfcode` field of the NAMESTR contains the formatting code for the variable. The example stores these codes in an array, `nfcode`, to use them later when the variables are printed.

---

* Do not attempt to use `nfl` to obtain the length of the format; use `nflen` instead.

SAS_XFFILE *continued*

```
long    *nfcodes,
        x;

ptr     infile,
        ptr_xv;

short   *list,
        nvar;

struct NAMESTR *namptr;

   /* access list and number of items in list */
list = (short *) SFLP(proc.head, 4);
nvar = SFN(proc.head, 4);

   /* allocate space to store format codes */
nfcodes = (long *)SAS_XMEMEX(nvar*sizeof(long));

   /* load formats */
if (SAS_XFFILE(infile, 'F', list, nvar)) {
   SAS_XPSLOG("%1z Could not load format.");
   SAS_XEXIT(XEXITWARNING, 0);
}

   /* define scatter reads */
SAS_XVGETI(infile, nvar, &ptr_xv);
for (x = 0; x < nvar; x++) {
   SAS_XVGETD(ptr_xv, list[x], . . .);
   SAS_XVNAME(infile, list[x], &namptr);
   nfcodes[x]=namptr->nfcode;
}
SAS_XVGETE(ptr_xv);

   /* do the scatter read */
while (SAS_XBYNEXT(infile) > 0) {
   while (SAS_XBYGET(infile) != NULL) {
      SAS_XVGET(ptr_xv, NULL);
      print the data using the formats stored
      in the nfcodes array
   }
}
```

SAS_XFNAME

Load informats and formats

Declarations

Type	Variable	Use	Description
ptr	*format-name*	input	pointer to an 8-byte name, padded with blanks if necessary
int	*format-type*	input	type of format or informat: 'F' = output format 'I' = informat
ptr	*ptr-to-fattrstr*	input	pointer to a format attribute structure (FATTRSTR): if you simply need to access the format code, use NULL if you need information on the minimum, maximum, and default width for the format, allocate the structure, set a pointer to the structure and then cast the pointer to type `ptr`
long	**ptr-to-iffcode*	output	pointer to the format or informat code to return
int	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc = SAS_XFNAME(format-name, format-type, ptr-to-fattrstr, ptr-to-iffcode);
```

Description

SAS_XFNAME is called by the SAS_XFFILE routine for every variable in the list. If you need to obtain formats for additional variables, such as a date-time format, you can call SAS_XFNAME directly. Refer to Chapter 13, "SAS Informats," and Chapter 14, "SAS Formats," in *SAS Language: Reference, Version 6, First Edition*, for a complete list of the SAS formats and informats. In addition to these, you can access any user-written formats available at your site.

If you want information on the minimum, maximum, or default lengths of the format, declare a copy of the format attribute structure and pass the address of the FATTRSTR to SAS_XFNAME. Normally, you do not need to know this information unless you are trying to validate the format specification. Refer to Chapter 20 for a complete description of the FATTRSTR.

SAS_XFNAME *continued*

Example

This example illustrates calling SAS_XFNAME to obtain the format code for a specific format, DATETIME. The example creates a new variable, called DTIME, stores the value returned by the call to SAS_ZDATTIM in it, and uses the format code to print the value of this variable. When the example finishes using the DATETIME format, it issues a call to SAS_XFDEL to delete the format code.

Note: The following example also illustrates using the address of a FATTRSTR in the call to SAS_XVNAME, although the example does not actually access any of the information in the FATTRSTR.

```
ptr     infile,
        outfile,
        out_ptr,
        ptr_xv;

short   nvar;

double  dtime;

long    iffcode;

char    tmp[20];

struct FATTRSTR tmp_fatt;
struct NAMESTR  tmp_namestr;

define scatter reads

   /* define gather writes  */
SAS_XVPUTI(outfile, nvar+1, &out_ptr);
calls to SAS_XVPUTD for output variables

   /* define the new variable  */
SAS_XVNAMEI(&tmp_namestr);
SAS_XFNAME("DATETIME", 'F', (ptr) &tmp_fatt, &iffcode);
tmp_namestr.nname = "DTIME";
tmp_namestr.namelen = 5;
memcpy(tmp_namestr.nform, "DATETIME", 8);
tmp_namestr.ntype = 1;
tmp_namestr.nlng = sizeof(double);
SAS_XVPUTD(out_ptr, &tmp_namestr, 0, &dtime, sizeof(double));
SAS_XVPUTE(out_ptr);

   /* do the scatter read and gather write  */
while (SAS_XBYNEXT(infile) > 0) {
   while (SAS_XBYGET(infile) != NULL) {
      SAS_XVGET(ptr_xv, NULL);
      process the input data
```

```
      /* get datetime value and format it for printing */
   dtime = SAS_ZDATTIM();
   SAS_XFXPN(dtime, 16, 0, iffcode, tmp);
   print the date to the print file or log

      /* gather and write the observation */
   SAS_XVPUT(out_ptr, NULL);
   SAS_XOADD(outfile, out_ptr);
 }
}
SAS_XFDEL(iffcode);
```

SAS_XFSHOW

Return name associated with a format or informat code

Declarations

Type	Variable	Use	Description
long	*iffcode*	input	format code to show values
ptr	*ptr-to-format*	returned	pointer to name of format

Usage

```
ptr-to-format = SAS_XFSHOW(iffcode);
```

Description

The SAS_XFSHOW routine allows you to determine what format or informat is associated with an access code. Access codes are assigned each time the procedure is executed. The primary use for this routine is to validate that the format code you are using is accessing the correct format.

If the returned name contains both @ and $, the format code refers to a character informat; names that contain only @ are for numeric informats. If the returned name contains only $, the code refers to a character format. When neither @ nor $ occur in the name, the code is for a numeric format.

SAS_XFSHOW *continued*

Example

The following example uses SAS_XFSHOW to determine if the format code for a variable has been correctly stored and accessed in the NAMESTR:

```
ptr     infile,
        outfile,
        ptr_xv,
        out_ptr;

short   *list,
        nvar;

int     differ,
        x;

ptr    tmpname;

struct NAMESTR *Nam_ptr;

   /* add format information to NAMESTRs */
SAS_XFFILE(infile, 'F', list, nvar);

   /* define scatter reads and gather writes */
SAS_XVGETI(infile, nvar, &ptr_xv);
SAS_XVPUTI(outfile, nvar+1, 0, &out_ptr);
for (x = 0; x < nvar; x++) {
   SAS_XVGETD(ptr_xv, list[x], ...);
   SAS_XVNAME(infile, list[x], &Nam_ptr);
   SAS_XVPUTD(out_ptr, Nam_ptr, ...);

   tmpname = SAS_XFSHOW((Nam_ptr)->nfcode);
   differ = SAS_ZSTRCOM((Nam_ptr)->nform, 8, tmpname, 8);
   if (differ) {
      perform error processing
   }
}
SAS_XVGETE(ptr_xv);
SAS_XVPUTE(out_ptr);
```

Note: This example works only for formats. If you want to use this example for an informat, you must begin comparing the second character of the `tmpname` variable with the `niform` field where the informat name is stored because the `niform` does not include the leading @.

SAS_XFXIC

Handle informatting requests for character values

Declarations

Type	**Variable**	**Use**	**Description**
ptr	*ptr-to-field*	input	pointer to informatting field.
int	*width*	input	width of input field.
long	*informat-code*	input	informat code obtained by calling XFNAME. Store this value in the `nicode` field of the NAMESTR for the variable. When *informat-code* is 0, the default informat is used.
ptr	*ptr-to-string*	output	pointer to resulting string.
int	*string-length*	input	length of output string.
int	*rc*	returned	return code: 0=successful nonzero = unsuccessful.

Usage

```
rc =  SAS_XFXIC(ptr-to-field, width, informat-code, ptr-to-string,
                string-length);
```

Description

This routine is useful when your procedure must read from an external file that has character data stored in special formats. You can access SAS character informats or informats that you have created with the FORMAT procedure or SAS/TOOLKIT software. The SAS_XFXIC routine translates a data value that is in a special informat to a character informat.

Example

This example shows how to read from an external file a value that is stored in hexadecimal characters and print the character representation of the value. You use the $HEX informat to read the data. The returned value has a length of 3.

```
#include <stdio>

long    iffcode;

char    in_line[8],
        frmttd_str[8];

FILE    *file_ptr;
```

SAS_XFXIC *continued*

```
   /* open an external file using the TESTIN fileref */
if ((file_ptr = open("TESTIN", "r")) == NULL) {
   SAS_XPSLOG("%1z In opening input text file");
   SAS_XEXIT(XEXITIO);
}

   /* read 7 characters from the external file */
fgets(in_line, 7, file_ptr);

   /* get the informat code and convert the input string to HEX */
SAS_XFNAME("$HEX    ", 'I', NULL, &iffcode);
SAS_XFXIC(in_line, strlen(in_line), iffcode, frmttd_str, 8);

   /* use the formatted string */
SAS_XPSLOG("in_line = %s%nfrmttd_str = %s", in_line, frmttd_str);

   /* close the external file */
fclose(file_ptr);
```

SAS_XFXIN

Handle informatting requests for numeric values

Declarations

Type	Variable	Use	Description
ptr	*ptr-to-field*	input	pointer to informatting field.
int	*width*	input	width of input field.
int	*number-decimals*	input	power of 10 to divide by.
long	*informat-code*	input	informat code obtained by calling SAS_XFNAME, which stores this value in the `nicode` field of the NAMESTR for the variable. When *informat-code* is 0, the default informat is used.
double	**ptr-to-value*	output	pointer to resulting double.
int	*rc*	returned	return code: 0=successful 2=reverted to BEST otherwise = unsuccessful.

Usage

```
rc =  SAS_XFXIN(ptr-to-field, width, number-decimals,informat-code,
                ptr-to-value);
```

Description

This routine is useful when your procedure must read numeric data from an external file. All data in external files are read as character data by your procedure. Even simple numeric strings such as '1234' must be converted to a numeric type. To convert the data to a numeric value, use either SAS_XFXIN or SAS_ZSTOL. Use the SAS_ZSTOL routine if you simply need to convert character data to a long integer. SAS_XFXIN converts character data to a floating-point value. The SAS_XFXIN routine can also translate a data value stored in a special informat to a floating-point value. You can access SAS numeric informats or informats that you have created with the FORMAT procedure or SAS/TOOLKIT software. This routine is useful for converting date values and packed numbers.

Example

This example illustrates how to read a value from an external file and convert it to a numeric value that can be printed or stored:

```
#include <stdio>

char    in_line[8];

double  in_num;

FILE    *file_ptr;

   /* open an external file using the TESTIN fileref */
if ((file_ptr = fopen("TESTIN", "r")) == NULL) {
   SAS_XPSLOG("%1z In opening input text file");
   SAS_XEXIT(XEXITIO);
}

   /* read 7 characters from the external file */
fgets(in_line, 7, file_ptr);

   /* convert the input string to a numeric value */
SAS_XFXIN(in_line, strlen(in_line), 0, 0, &in_num);

   /* use the formatted string */
SAS_XPSLOG("in_line = %s%nin_num = %f", in_line, in_num);

   /* close the external file */
fclose(file_ptr);
```

SAS_XFXPC

Handle formatting requests for character values

Declarations

Type	Variable	Use	Description
ptr	*ptr-to-char-value*	input	pointer to string to format
int	*width*	input	width of input field
long	*format-code*	input	format code to use. If *format-code* is 0, the default format ($CHAR*w*.) is used.
ptr	*ptr-to-string*	output	pointer to resulting string
int	*string-length*	input	length of output string
int	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc =  SAS_XFXPC(ptr-to-char-value, width, format-code, ptr-to-string,
               string-length);
```

Description

If you want to format a character value into a different character string, use SAS_XFXPC. In most cases, before you call SAS_XFXPC, you need to call SAS_XFNAME to load the format and obtain the format code.

If you want to format a character value using $CHAR*w*., you do not need to call SAS_XFNAME and get a format code. Use the format code 0 for the $CHAR*w*. format. Note that if you are using the $CHAR format, you are actually just moving character data, not formatting it. In this case, it is more efficient to simply move data from one location to another using SAS_ZMOVEI.

The SAS_XFXPC routine is most useful when you are formatting with character formats such as $HEX*w*. or user-written formats that perform table lookup. These formats actually change the data when formatting.

You must ensure that the character string where the output value is to be stored is large enough to hold the value. SAS_XFXPC has no way of knowing the size of this target. SAS_XFXPC also does not pad to the size of the target, but it will pad to the field width you specify.

Example

This example shows how to call SAS_XFNAME to get the format code for the $HEX format and then call SAS_XFXPC to format a character string into its hexadecimal equivalent.

```
long    iffcode;
char    input_str[6] = "123456",
        formatted_str[30];

   /* load the format code */
SAS_XFNAME("$HEX    ", 'F', NULL, &iffcode);

   /* format the character value */
SAS_XFXPC(input_str, 6, iffcode, formatted_str, 30);

   /* the formatted string is ready to use */
SAS_XPSLOG("input string = %s%nformatted string = %s", input_str,
           formatted_str);
```

SAS_XFXPN

Handle formatting requests for numeric values

Declarations

Type	Variable	Use	Description
double	*value*	input	input value
int	*width*	input	width of output field
int	*number-decimals*	input	power of 10 to multiply by
long	*format-code*	input	format code to use. If *format-code* is 0, the default format (*w.d*) is used.
ptr	*ptr-to-value*	output	pointer to resulting string
int	*rc*	returned	return code: 0 = successful; nonzero = unsuccessful

Usage

```
rc = SAS_XFXPN(value, width, number-decimals, format-code, ptr-to-value);
```

Description

If you want to format a numeric value stored as a double into a character string, use SAS_XFXPN. In most cases, before you call SAS_XFXPN, you need to call SAS_XFNAME to load the format and obtain the format code.

Note: If you want to format an integer into a character string, use the SAS_ZLTOS routine, described in Chapter 23, "SAS_Z Routines."

SAS_XFXPN *continued*

If you want to format a number using *w.d* (for example, 10.), it is not necessary to call SAS_XFNAME to get a format code. Use the format code 0 for the *w.d* numeric format.

You must ensure that the character string where the output value is to be stored is large enough to hold the value. SAS_XFXPN has no way of knowing the size of this target. SAS_XFXPN also does not pad to the size of the target, but it will pad to the field width you specify.

Example

Suppose you want to format the value 1992 in the ROMAN format into an 11-byte character string. First call SAS_XFNAME to load the ROMAN format and obtain the format code, then call SAS_XFXPN to format the value:

```
long   iffcode;
double value;
char   string[11];

   /* load the format code */
SAS_XFNAME("ROMAN    ", 'F', NULL, &iffcode);

value = 1992;
SAS_XFXPN(value, 11, 0, iffcode, string);

   /* the formatted string is ready to use */
SAS_XPSLOG("input value = %f%nformatted string = %s", value,
           string);
```

The character array *string* has the value MCMXCII.

SAS_XM Routines: Memory Management

The SAS_XM routines allow your procedure to allocate memory as it is needed. All applications should acquire memory using these routines instead of using facilities supplied by the language (such as `malloc` and `free` in C, or ALLOCATE and FREE in PL/I). If you use the language's memory management facilities, you may acquire memory that fragments the pools of memory available to the SAS System.

Note: If you are writing your procedure in the FORTRAN language, you need to read the information on working with pointers in Chapter 26, "Writing Programs in the FORTRAN Language," before attempting to use the SAS_XM routines.

General Description of Memory Allocation

Before using any of these routines, you need to know how to calculate storage requirements. This process is described in the "Allocating Memory" section of Chapter 10, "Writing a SAS Procedure."

All memory is acquired from memory pools, which contain chunks of memory in an efficient size for your host system. The memory allocation routines return pointers to the memory. If you call a memory allocation routine and the value returned by the routine is NULL, the memory pool available to your procedure does not contain enough memory to satisfy your procedure's request. If the routine returns a non-NULL value, the memory is available. In this case, your procedure must cast the returned pointer to the correct type of pointer before using the allocated space.

Note: In order to test the value returned by the memory allocation routine against the value of a NULL pointer, you need to know the correct value of NULL.

- In the C language, the symbol NULL refers to a null pointer, which has the value of 0.
- In the FORTRAN language, NULL is equivalent to an INTEGER*4 value of 0.
- In the PL/I language, you should call the SAS_NUL routine to obtain a true null pointer. Using the built-in function NULL may give you a pointer that is not entirely 0.
- In the IBM 370 assember language, a 4-byte integer 0 aligned on a fullword boundary is a null pointer. Either of the following declarations is valid:

```
NULL     DC     F'0'
NULL     DC     A(0)
```

All memory acquired by the SAS_XM routines is freed at the end of your procedure. Therefore, you do not need to explicitly free everything that you allocate. It is a good idea (and often necessary), however, to free memory when it is no longer needed in order to allow the memory to be reused by later portions of your procedure.

Comparison of Memory Allocation Techniques

SAS/TOOLKIT software provides two sets of routines for allocating memory: the SAS_XMEM routines and the SAS_XMPOOL routines. You cannot intermix these routines for a specific block of memory; although it is possible to use both types of routines in one procedure, they must be working with separate areas of memory. This is especially true when you need to free memory.

Both sets of routines work with the SAS Supervisor to manage space, but they handle allocations differently. The SAS_XMEM routines allocate space from a standard system allocation pool. The SAS_XMPOOL*x*, SAS_XMFREE, and SAS_XMALLOC routines allocate and manage a separate pool of space.

Using the SAS_XMEM Routines

The primary reason for using the SAS_XMEM routines is that you can allocate memory with a single call and these routines require fewer parameters. If your procedure needs to allocate a few large blocks of memory, the SAS_XMEM routines would probably be a better choice.

Use the SAS_XMEM routines as follows:

- Allocate memory with SAS_XMEMGET, SAS_XMEMZER, or SAS_XMEMEX. SAS_XMEMGET simply attempts to allocate memory, takes no action (except to return a NULL pointer) if the allocation fails, and does not initialize the allocated memory if it is successful. SAS_XMEMZER attempts to allocate memory, takes no action (except to return a NULL pointer) if the allocation fails, and sets the memory to 0s if the allocation is successful. SAS_XMEMEX attempts to allocate, sets memory to 0 if it is successful, and causes the procedure to end if the allocation is unsuccessful.
- Free the allocated memory with SAS_XMEMFRE.

Using the SAS_XMALLOC and SAS_XMPOOL Routines

In general, if your procedure needs to allocate numerous small blocks of memory, you should use the SAS_XMPOOL*x* routines. These routines provide a large block of memory that can be apportioned into smaller blocks for individual lists. The primary reason for using the SAS_XMALLOC and SAS_XMPOOL routines is that you can free all the individual memory allocations made by calls to SAS_XMALLOC with a single call to SAS_XMFREE. If your program needs to allocate numerous areas of memory and then free those areas later in the program, you should probably use SAS_XMPOOL and associated routines.

Use the SAS_XMALLOC and SAS_XMPOOL routines as follows:

- Create a pool for your program to use by calling SAS_XMPOOLC. If you want to use the standard procedure memory pool, omit this step.
- Use portions of the allocated pool by calling SAS_XMALLOC. If you are using the standard memory pool, use XM_HEAP in the call to SAS_XMALLOC instead of the pointer returned by SAS_XMALLOC.
- Free portions of the pool by calling SAS_XMFREE. Note that these calls free individual portions that were allocated by calling SAS_XMALLOC.
- Delete the entire pool allocated with SAS_XMPOOLC by calling SAS_XMPOOLD. Do not call SAS_XMPOOLD if your procedure uses the standard procedure pool, XM_HEAP.

SAS_XMALLOC

Allocate memory from a given pool

Declarations

Type	Variable	Use	Description
ptr	*pool-id-ptr*	input	pool ID pointer. Use either of the following: pointer returned by the call to SAS_XMPOOLC XM_HEAP, to access the standard procedure memory
long	*length*	input	amount of memory in bytes.
int	*flags*	input	memory flags. You can add any combination of these flags: XM_EXIT XM_SIZE XM_ZERO.
ptr	*pool-part-ptr*	returned	pointer to memory allocated.

Usage

```
pool-part-ptr = SAS_XMALLOC(pool-id-ptr,length, flags);
```

Description

Before you call SAS_XMALLOC, you must first call SAS_XMPOOLC. The call to SAS_XMPOOLC allocates a memory pool for use by your procedure. Each time your procedure needs memory space to store data, call SAS_XMALLOC to access a portion of the memory obtained by SAS_XMPOOLC.

In the call to SAS_XMALLOC, you can specify flags to indicate any or all of the following:

XM_EXIT — exits from the procedure if the memory space is not available.

XM_SIZE — keeps track of the size of the space allocated so that space can later be freed. If you specify this flag, SAS_XMALLOC stores the size and other routines access the size so that you do not need to specify it in later calls to SAS_XM routines.

XM_ZERO — sets the allocated memory to 0.

You combine these flags by adding the flags together (in C, you can also combine flags with the OR operator (|)). You can also specify these flags in the call to SAS_XMPOOLC to affect the entire pool of memory. If you specify some, but not all, of these flags in the call to SAS_XMPOOLC, you can specify additional ones in the call to SAS_XMALLOC. In that case, the memory accessed by the

SAS_XMALLOC *continued*

SAS_XMALLOC call has the combined effect of the flags specified by the call to SAS_XMALLOC as well as the flags set for the entire pool.

If you want to free memory allocated by SAS_XMALLOC, use SAS_XMFREE. You can also free the entire pool allocated by SAS_XMPOOLC by calling SAS_XMPOOLD. Keep in mind that you cannot access memory after you have freed it.

▶ ***Caution*** .... ***Do not call SAS_XMPOOLD for any pool that you did not create, such as the SAS System memory pool, XM_HEAP.*** ▲

Example

In this example, SAS_XMALLOC obtains a portion of memory to store an array of doubles.

```
short   nvar;
double *varptr;

nvar = SFN(proc.head, 4);
varptr = (double *)SAS_XMALLOC(XM_HEAP, sizeof(double)*nvar,
        (XM_EXIT+XM_SIZE+XM_ZERO));
```

Note that the example

- □ uses the standard procedure pool of memory, XM_HEAP, so it does not need to call SAS_XMPOOLC to create a pool of memory for calls to SAS_XMALLOC.
- □ sets all three flags so the size of the allocated space is noted for later use, the memory is filled with 0s, and the procedure exits if the memory is not available.
- □ casts the pointer returned by SAS_XMALLOC to the appropriate type of pointer.

SAS_XMEMEX

Allocate memory, exiting the procedure if unavailable

Declarations

Type	Variable	Use	Description
long	*length*	input	amount of memory to allocate
ptr	*memory-ptr*	returned	pointer to memory

Usage

```
memory-ptr = SAS_XMEMEX(length);
```

Description

This routine allocates memory from a standard memory pool available to the SAS System. SAS_XMEMEX sets the memory to 0 and keeps track of the length so a subsequent call to SAS_XMEMFRE can free the memory. If you want to free memory allocated by SAS_XMEMEX after your procedure has finished using it, you must call SAS_XMEMFRE, but it is not necessary to free memory in your procedure. It is freed automatically at step termination.

If the amount of memory requested by your procedure is not available, the routine issues a message saying the memory is unavailable and it calls SAS_XEXIT to stop the procedure.

Example

In this example, SAS_XMEMEX allocates memory for an array of doubles. Note that the pointer returned by SAS_XMEMEX must be cast to the appropriate type of pointer. You do not need to test the pointer returned by SAS_XMEMEX because the routine automatically causes the program to end if the memory is not available.

```
short   nvar;
double *varptr;

nvar = SFN(proc.head, 4);
varptr = (double *)SAS_XMEMEX(sizeof(double)*nvar);
```

SAS_XMEMFRE

Free memory allocated by SAS_XMEMGET, SAS_XMEMZER, or SAS_XMEMEX

Declarations

Type	Variable	Use	Description
ptr	*memory-ptr*	input	pointer to memory to free

Usage

C	`SAS_XMEMFRE(memory-ptr);`
FORTRAN	`CALL SAS_XMEMFRE(memory-ptr)`
PL/I	`CALL SAS_XMEMFRE(memory-ptr);`

SAS_XMEMFRE *continued*

Description

SAS_XMEMFRE frees memory allocated with SAS_XMEMGET, SAS_XMEMZER, or SAS_XMEMEX. Do not attempt to use a pointer that was not returned by one of these routines. Calling SAS_XMEMFRE with a NULL pointer has no effect.

If you do not explicitly free memory, it is automatically freed when your procedure ends. Thus, you only need to use this routine if you must free memory while your program is executing.

▶ *Caution* .... ***Do not attempt to use the space pointed to by memory-ptr after you call this routine.*** It is a good idea to set this pointer to NULL after you call SAS_XMEMFRE. ▲

Example

This example allocates an array of doubles, which is used by the program. When the program has finished processing the array, the space is freed. Note that `varptr` is reset to NULL to avoid inadvertent attempts to reuse the space.

```
short   nvar;
double *varptr;

nvar = SFN(proc.head, 4);
varptr = (double *)SAS_XMEMZER(sizeof(double)*nvar);
if (varptr == NULL) {
   SAS_XEXIT(XEXITMEMORY, 0);
}
program processes array

   /* free memory after processing is completed */
SAS_XMEMFRE(varptr);
varptr = NULL;
```

SAS_XMEMGET

Allocate memory

Declarations

Type	Variable	Use	Description
long	*length*	input	amount of memory
ptr	*memory-ptr*	returned	pointer to allocated memory NULL = memory not successfully allocated

Usage

```
memory-ptr = SAS_XMEMGET(length);
```

Description

The SAS_XMEMGET routine attempts to allocate the requested amount of memory. If this amount is not available, the routine returns a null pointer. If the memory is available, the routine allocates it for the calling program but does not change the contents of the memory; the SAS_XMEMZER routine is the same as this routine except SAS_XMEMZER sets the memory contents to 0.

▶ *Caution* .... ***You must check to ensure that the SAS_XMEMGET routine returns a non-NULL pointer before using the pointer.*** ▲

Example

This example allocates memory for an array of doubles. Note that the pointer returned from SAS_XMEMGET must be cast to the appropriate type of pointer. When you use SAS_XMEMGET, be sure to test the pointer returned by the routine to see if the memory allocation was successful.

```
short   nvar;
double *varptr;

nvar = SFN(proc.head, 4);
varptr = (double *)SAS_XMEMGET(sizeof(double)*nvar);
if (varptr == NULL) {
   SAS_XEXIT(XEXITMEMORY, 0);
}
```

SAS_XMEMZER

Allocate memory and zero it

Declarations

Type	**Variable**	**Use**	**Description**
long	*length*	input	amount of memory
ptr	*memory-ptr*	returned	pointer to allocated memory NULL = memory not successfully allocated

SAS_XMEMZER *continued*

Usage

```
memory-ptr = SAS_XMEMZER(length);
```

Description

The SAS_XMEMZER routine attempts to allocate the requested amount of memory. If this amount is not available, the routine returns a null pointer. If the memory is available, the routine allocates it and fills the area with 0s.

Example

This example allocates memory for an array of doubles and sets the areas to 0s. Note that the pointer returned from SAS_XMEMZER must be cast to the appropriate type of pointer. When you use SAS_XMEMZER, be sure to test the pointer returned by the routine to see if the memory allocation was successful.

```
short   nvar;
double *varptr;

nvar = SFN(proc.head, 4);
varptr = (double *)SAS_XMEMZER(sizeof(double)*nvar);
if (varptr == NULL) {
   SAS_XEXIT(XEXITMEMORY, 0);
}
```

SAS_XMFREE

Free memory within a pool

Declarations

Type	Variable	Use	Description
ptr	*pool-id-ptr*	input	pool ID
ptr	*pool-part-ptr*	input	pointer to memory to free
long	*length*	input	amount of memory to free in bytes −1 = XM_SIZE specified when memory was allocated

Usage

C	`SAS_XMFREE(pool-id-ptr, pool-part-ptr, length);`
FORTRAN	`CALL SAS_XMFREE(pool-id-ptr, pool-part-ptr, length)`
PL/I	`CALL SAS_XMFREE(pool-id-ptr, pool-part-ptr, length);`

Description

Use this routine to free memory accessed with SAS_XMALLOC. You can only specify a −1 for *length* if you specified the XM_SIZE flag when the memory was allocated with SAS_XMPOOLC or accessed with SAS_XMALLOC.

SAS_XMFREE can be used only to free memory accessed with SAS_XMALLOC. To free an entire pool of memory allocated by SAS_XMPOOLC, use SAS_XMPOOLD. To free memory allocated with any of the SAS_XMEM routines, use SAS_XMEMFRE.

Example

This example frees a portion of memory allocated by a call to SAS_XMALLOC. This example includes the following steps:

- SAS_XMPOOLC allocates a standard pool of memory and sets two flags. These flags indicate that the program should exit if the memory is not available and that if memory is available, the program should keep track of the size of each portion obtained by subsequent calls to SAS_XMALLOC.
- The two calls to SAS_XMALLOC obtain from the memory pool allocated by SAS_XMPOOLC enough memory to store an array of doubles and a copy of the XOPNSTR structure. Note that in both cases, the pointer returned by SAS_XMALLOC is cast to the appropriate pointer.
- After the program finishes using the XOPNSTR, it calls SAS_XMFREE and passes the pointer to the XOPNSTR. Only the memory allocated for storing the XOPNSTR is freed. Since the earlier call to SAS_XMPOOLC set the XM_SIZE flag so that the size of each block of allocate memory is noted, you can set the *length* parameter in SAS_XMFREE to −1. Note that in the call to SAS_XMFREE, you must recast the pointer for the allocated space back to `ptr`.

The example follows.

```
short   nvar;
double  *varptr;
struct  XOPNSTR *opstptr;
ptr     poolptr;

   /* create a pool of memory using default sizes */
poolptr = SAS_XMPOOLC(XM_ISA, XM_OSA, (XM_EXIT+XM_SIZE));

   /* allocate an array of doubles */
nvar = SFN(proc.head, 4);
varptr = (double *)SAS_XMALLOC(poolptr, sizeof(double)*nvar);
```

SAS_XMFREE *continued*

```
   /* allocate space for an XOPNSTR */
opstptr = (struct XOPNSTR *)SAS_XMALLOC(poolptr, sizeof(struct XOPNSTR));
program uses XOPNSTR

   /* free space for XOPNSTR */
SAS_XMFREE(poolptr, (ptr)opstptr, -1);
```

SAS_XMPOOLC

Create a new pool

Declarations

Type	Variable	Use	Description
long	*initial-size*	input	initial storage allocation: actual space needed XM_ISA = default storage size.
long	*overflow-size*	input	overflow storage allocation: actual space needed 0 = disallow overflow allocations XM_OSA = default storage size.
int	*flags*	input	flags to indicate how to handle allocated memory. Refer to “Description” for a description of each of these flags. One or any combination of the following are permitted: XM_EXIT XM_SIZE XM_ZERO.
ptr	*pool-id-ptr*	returned	pool ID.

Usage

```
pool-id-ptr = SAS_XMPOOLC(initial-size, overflow-size, flags);
```

Description

If you want to allocate and manage your own pool of memory in your procedure, call SAS_XMPOOLC once to set up the memory pool. Then you can place multiple calls to SAS_XMALLOC to access the memory in this pool.

Note: You do not need to create your own pool of memory; you can allocate memory from the standard pool by calling the SAS_XMEM routines or by calling SAS_XMALLOC with XM_HEAP for the pool ID.

When you create a pool of memory, you have several options:

- You can specify the actual quantity of space you will need in the *initial-size* parameter.
- You can specify the overflow storage needs in the *over-flow* parameter or disallow overflow by specifying 0.
- You can use the default space allocations, which differ on different host systems, for either initial (XM_ISA) or overflow (XM_OSA) allocations.

If you want to free memory allocated by SAS_XMPOOLC, use the SAS_XMPOOLD routine to free the entire pool. Use SAS_XMFREE to free portions of the pool that were accessed by calls to SAS_XMALLOC.

In the call to SAS_XMPOOLC, you can specify flags to indicate any or all of the following:

XM_EXIT	exits from the procedure if the memory space is not available.
XM_SIZE	keeps track of the size of the space allocated so that space can later be freed.
XM_ZERO	sets the allocated memory to 0.

You can combine these flags by adding the flags together (in C, you can also combine flags with the OR operator (|)). When you specify these flags in the call to SAS_XMPOOLC, they affect the entire pool of memory. You can also specify these flags in the call to SAS_XMALLOC to affect limited portions of the pool. Refer to the discussion of SAS_XMALLOC for more information.

Example

Refer to the example for SAS_XMFREE for an illustration of this routine.

SAS_XMPOOLD

Delete a pool

Declarations

Type	Variable	Use	Description
ptr	*pool-id-ptr*	input	pool ID of pool to delete

Usage

C	`SAS_XMPOOLD(pool-id-ptr);`
FORTRAN	`CALL SAS_XMPOOLD(pool-id-ptr)`
PL/I	`CALL SAS_XMPOOLD(pool-id-ptr);`

SAS_XMPOOLD *continued*

Description

The SAS_XMPOOLD routine frees the entire pool of storage allocated for your procedure. You can also free separate portions of the pool that were accessed by calls to SAS_XMALLOC by using the SAS_XMFREE routine. It is not necessary, however, to free individual portions of the pool before calling SAS_XMPOOLD.

The pool is automatically freed when your procedure terminates, so it is not necessary to call this routine.

▶ ***Caution*** .... ***Do not use XM_HEAP for the pool-id-ptr.***
Deleting the SAS System memory pool causes the SAS System to abend. ▲

Example

This example allocates two portions of memory with SAS_XMALLOC and frees both with a single call to SAS_XMPOOLD. This example includes the following steps:

- SAS_XMPOOLC allocates a standard pool of memory and sets the XM_EXIT flag to indicate that the program should exit if the requested memory is not available.
- The two calls to SAS_XMALLOC obtain from the memory pool allocated by SAS_XMPOOLC enough memory to store an array of doubles and a copy of the XOPNSTR structure. Note that in both cases, the pointer returned by SAS_XMALLOC is cast to the appropriate pointer.
- After the program finishes using both the XOPNSTR and the variable array, it calls SAS_XMPOOLD to free all of the space allocated by SAS_XMPOOLC.

The example follows.

```
short   nvar;
double  *varptr;
struct  XOPNSTR *opstptr;
ptr     poolptr;


   /* create a pool of memory using default sizes */
poolptr = SAS_XMPOOLC(XM_ISA, XM_OSA, XM_EXIT);

   /* allocate an array of doubles */
nvar = SFN(proc.head, 4);
varptr = (double *)SAS_XMALLOC(poolptr, sizeof(double)*nvar);

   /* allocate space for an XOPNSTR */
opstptr = (struct XOPNSTR *)SAS_XMALLOC(poolptr, sizeof(struct XOPNSTR));
program processes XOPNSTR and variable array

   /* free all the space in the pool */
SAS_XMPOOLD(poolptr);
```

SAS_XO Routines: Observation Level

The SAS_XO routines handle processing of SAS data sets on the data set and observation level. Use SAS_XO routines to perform the following tasks:

- open a SAS data set
- close a SAS data set
- obtain descriptive information about the data set
- read observations without regard to BY processing
- write records to output SAS data sets
- move to a specific position in the SAS data set.

Note: All of the SAS_XO routines use a *file-ptr* argument to identify the file to be processed. Refer to "Opening SAS Data Sets" in Chapter 10 for information on extracting file references from the statement structure.

What Is a SAS Data Set?

A SAS data set is a collection of data values and their associated descriptive information arranged and presented in a form that can be recognized and processed by the SAS System. All SAS data sets contain these two logical components:

- data values organized into a rectangular structure of columns and rows
- descriptor information that identifies attributes of both the data set and the data values.

You can visualize a SAS data set as a table where the variables make up the columns and the observations make up the rows. The SAS_XO routines work with the descriptive information in a SAS data set to maneuver through the data set and to provide information about the data set.

Not all SAS data sets have the same processing characteristics. The processing characteristics of a data set are determined by the engine used to access the data set. An *engine* is the set of instructions the SAS System uses to read from and write to a file. The SAS System provides engines to read many kinds of files, including the following:

- SAS data sets created by Release 6.06 and later of the SAS System
- SAS data sets created by Version 5 (and Release 82.4) of the SAS System
- SAS data sets in transport format
- SAS data sets stored on tape or in sequential format on disk

- files formatted by other software, such as SPSS, BMDP, and OSIRIS.*
- views created by the SQL procedure or SAS/ACCESS software.

You specify the capabilities that the engine must support when you open the data set. Opening a SAS data set is discussed in the next section. Refer to "Data Set Engines" later in this chapter for more information on engines and their capabilities.

Opening a SAS Data Set

User-written procedures can open a SAS data set by one of two methods:

- by using the @DS and @DSDFLT semantic actions in the grammar to open a data set specified when the procedure is invoked. This is the most common method of opening a data set. Refer to Chapter 3, "Writing a Grammar," for more information on these semantic actions. If you use this method to open a SAS data set, you do not need to call SAS_XOOPEN for that data set.
- by calling SAS_XOOPEN to open a data set from within the procedure. This process allows your procedure to determine if it needs a data set before opening it. Note that if you open a data set using this process, you cannot specify record-level access to the data set. See "Levels of Access" later in this chapter.

When you open a data set, you specify the kinds of capabilites the engine that accesses the data set must have for your procedure to be able to process the data set. For example, if your procedure needs to be able to have random access to the observations in the data set, the parameters to the @DS semantic action must specify that need. If the user attempts to specify a data set that has an engine that does not support the capabilities required by your procedure, the user receives the following message:

```
ERROR: Attempted function cannot be performed for engine type and data set name.
```

▶ *Caution* .... ***If your procedure uses routines that work only for certain engines, be sure your grammar specifies the appropriate parameters to the @DS semantic action that opens the data set.***
If you fail to specify the limitations on data sets in the grammar, the results of using an inappropriate data set with a procedure that requires special engine capabilities are unpredictable. ▲

Levels of Access

In most cases, when you open a data set, your procedure has *member-level access* to the data set; that is, no other process can make changes to the data set until your procedure completes. If your procedure will be used with engines that support *record-level locking* (such as SAS/SHARE software provides), you can

* OSIRIS was produced by the Institute for Social Research, Ann Arbor, Michigan.

specify when you open the data set that you want to have only record-level access to the data set. Record-level access means that your procedure locks access to only the observation that it is currently processing. You should not specify record-level access to a data set if your procedure ever rewinds the data set or rereads observations, because the contents of an observation could change between the times that your procedure reads it if you have only record-level access.

You specify record-level access or member-level access in the sixth parameter to the @DS or @DSDFLT semantic action that opens the data set. Refer to Chapter 6, "SAS Files," in *SAS Language: Reference, Version 6, First Edition* for more information on levels of locking.

Observation Numbers and Record IDs

In the SAS System, the observation number appears in many forms of SAS output and indicates the relative position of an observation in the data set when you read the entire data set. The observation number is not a permanent value stored in an observation; it is generated by the procedure that produces the output.

If you are reading a data set sequentially, you do not need to have the observation number to access an observation. You simply start at the beginning of the data set and read through it. Although you do not access the observation number in your procedure, you may want to print the observation number in the output that your procedure creates. For sequential data sets, the simplest method for generating observation numbers is to create a counter that you increment each time you read an observation from the data set.

If you are accessing a data set randomly, or if the input data set is subsetted by the use of a FIRSTOBS= SAS system option or a WHERE clause, you need to obtain the observation number differently. Instead of generating it, you need to transform the internal record identifier, called the record ID, to the observation number that shows the true position of the observation in the data set. The *record ID* is a unique identifier for each observation in a data set. The format for the record ID differs depending on the engine that accesses the file and for many engines, the record ID is only a temporary identifier for the observation.

Use the SAS_XONOTE and SAS_XORID2N routines to obtain the record ID and transform it to the observation number. Note that when your procedure uses these two routines, you limit the data set engines that can be used with your procedure to only those that are radix addressable (that is, they must be capable of transforming record IDs into observation numbers).

If you are reading a data set using random access (for example, you want to return to an observation that you previously read), you must use the record ID to locate the observation. The observation number is not sufficient information to enable an engine to locate an observation.

Data Set Engines

Some of the capabilities of SAS/TOOLKIT software cannot be used with some engines. In particular, the SAS_XO routines only work when the engine supports the function of the SAS_XO routine. For example, a sequential file cannot be randomly accessed. A sequential file is usually stored on tape or another device that does not support random access. You can read from or write to a sequential file from start to finish, rewind the file and then read it again. You cannot, however, randomly process the observations in a sequential file.

As you write your procedure, you need to consider what kind of SAS data sets will be used with your procedure. If your procedure is designed to process a specific SAS data set, you can determine the capabilities of the engine that supports that data set and write your procedure to take advantage of the full capabilities of the engine. If, however, your procedure is a more general procedure, like many of the procedures provided with base SAS software, you should use only the SAS_X routines that function with all or most of the engines.

Table 22.1 summarizes the capabilities of the most commonly available engines.

Table 22.1 *Engine Capabilities*

		Capabilities Supported				
Engine Name	**Hosts Where Available**	**Output Mode**	**Random Access**	**Record-level Locking**	**Rewind BY Groups**	**Radix Addressable**
BASE, V606, or V607	all	yes	yes	yes	yes	yes
BASE with compression	all	yes	yes	yes	yes	no
BMDP	CMS, MVS	no	no	no	yes	no
OSIRIS	all	no	no	no	yes	no
SASIODB2	MVS	yes	no	no	yes	no
SASIODSR	VMS	yes	yes	no	yes	yes
SASIOSQD	CMS	yes	no	no	yes	no
SASIOS2K	CMS, MVS	yes	yes	no	yes	yes
SQLVIEW	all	no	yes	no	yes	yes
SPSS	all	no	no	no	yes	no
TAPE	all	yes	no	no	yes*	yes
V5	CMS, MVS, VMS	yes	yes	no	yes	yes
V5SEQ	CMS, MVS	yes	no	no	yes*	no
XPORT	all	yes	no	no	yes	no

* The BY groups in V5SEQ or TAPE files cannot be rewound across volumes. Therefore, this capability is available only when the data set is stored on a single volume.

Table 22.2 lists the routines that you can use only when the data set engine supports the capabilities listed in Table 22.1.

Table 22.2 SAS_X Routines and Required Engine Capabilities

SAS_X Routine	Required Engine Capability
SAS_XBYRWND(*fileid*,'B')	rewind BY group. In addition, you should set the *access-1* and *access-2* parameters of the @DS semantic action to 2 to avoid changes to an observation while your procedure is in the middle of processing a BY group.
SAS_XOADD	output mode
SAS_XONOTE	radix addressable
SAS_XON2RID	radix addressable
SAS_XOPNT	random access, radix addressable
SAS_XORID2N	radix addressable

SAS_XOADD

Add a record to an open data set

Declarations

Type	Variable	Use	Description
ptr	*output-file-ptr*	input	fileid of an open output data set
ptr	*record-ptr*	input	record to be added: specify NULL to use the buffer used by SAS_XVPUT when building the observation
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc = SAS_XOADD(output-file-ptr, record-ptr);
```

Description

This function adds a new observation to a SAS data set. If your procedure creates a SAS data set, you use the SAS_XVPUT routines to build the observation, and then you call SAS_XOADD to write the observation to the data set.

This routine can be called for files opened for output. If you specify NULL for the *record-ptr* argument, the buffer used by SAS_XVPUT to build the observation is used by SAS_XOADD.

SAS_XOADD *continued*

Example

This example illustrates how to add records to a SAS data set. It shows the loop to read from one data set, write to another, and then close the input data set.

```
ptr  infile,
     ptr_xv,
     out_ptr,
     outfile;

define scatter reads and gather writes

   /* read and write observations */
while (SAS_XBYNEXT(infile) > 0) {
   while (SAS_XBYGET(infile) != NULL) {
      SAS_XVGET(ptr_xv, NULL);
      process input file
      SAS_XVPUT(out_ptr, NULL);

         /* write observation to output data set */
      SAS_XOADD(outfile, NULL);
   }
}
SAS_XOCLOSE(infile, 0);
perform further processing on output file
SAS_EXIT(XEXITNORMAL, 0);
```

SAS_XOCLOSE

Close a SAS data set

Declarations

Type	Variable	Use	Description
ptr	*file-ptr*	input	fileid of an open data set
int	*disposition*	input	file disposition: 0 = simple close of the file XO_DELETE = delete this file after close
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc = SAS_XOCLOSE(file-ptr, disposition);
```

Description

You can use SAS_XOCLOSE to close a SAS data set, but it is not necessary to call SAS_XOCLOSE. When you call SAS_XOCLOSE for an output data set, it writes a message in the log reporting the number of variables and observations.

All opened SAS data sets are closed automatically when the procedure terminates with SAS_XEXIT. Output data sets closed by SAS_XEXIT replace existing files of the same name (unless the user has specified the NOREPLACE SAS system option or data set option).

If you use SAS_XOCLOSE to close a data set, you must specify whether the data set should be saved or not. The valid values for the *disposition* argument are described below:

0	closes the file and, if the file was opened for output, replaces an existing file of the same name.
XO_DELETE	deletes the file being closed. If the file was opened for output, then the file is deleted without replacing an existing file of the same name.

Example

Refer to the example for SAS_XOADD for an illustration of using SAS_XOCLOSE.

SAS_XOGETN

Read observations when no BY statement possible

Declarations

Type	Variable	Use	Description
ptr	*file-ptr*	input	fileid of an open input data set
ptr	**ptr-to-record-ptr*	update	pointer to the record that is read: set this to NULL if you are using SAS_XVGET to process the observation
rctype	*rc*	returned	return code: 0 = successful X_WEOF = end of file encountered otherwise = unsuccessful

SAS_XOGETN *continued*

Usage

```
rc = SAS_XOGETN(file-ptr, ptr-to-record-ptr);
```

Description

This routine returns the next observation in a SAS data set. The next record is defined as the record that follows the last record read by SAS_XOGETN or as the record positioned to by SAS_XOPNT.

▶ *Caution* .... ***Use this routine to read observations only if your procedure does not permit BY statements.***

If your procedure permits a BY statement, always use the SAS_XBYGET routine to read from the data set. ▲

When the user does not specify the BY statement, the SAS_XBYGET routine reads the data set without regard to BY groups. You can use SAS_XBYGET to read a data set even when your procedure does not permit the use of a BY statement, but SAS_XOGETN may provide faster execution for procedures that do not permit BY statements.

You cannot use SAS_XOGETN with a data set opened for output.

If the user specifies a value for the FIRSTOBS= or OBS= SAS system options, SAS_XOGETN recognizes these boundaries. Your procedure does not need to test the observation number to ensure that it is within these boundaries.

Example

This example illustrates reading from an input file when no BY statement is permitted by the procedure.

```
ptr  infile,
     ptr_xv,
     out_ptr,
     outfile;

   /* read observations when no BY statement is permitted */
while (SAS_XOGETN(infile, NULL) == 0) {
   SAS_XVGET(ptr_xv, NULL);
   process input file
   SAS_XVPUT(out_ptr, NULL);

      /* write observation to output data set */
   SAS_XOADD(outfile, NULL);
}
SAS_XEXIT(XEXITNORMAL, 0);
```

SAS_XOINFO

Inquire about or set header information in an opened file

Declarations

Type	Variable	Use	Description
ptr	*file-ptr*	input	fileid of an open data set
int	*flag*	input	action flag
varies	*storage*	update	data field for storing information. See Tables 22.3 and 22.4.
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc = SAS_XOINFO(file-ptr, flag, (ptr) &storage);
```

Description

SAS_XOINFO allows you to obtain information about an existing data set and set information when you are creating an output data set. The parameters you specify to SAS_XOINFO determine the kind of information it returns or the actions it takes. The first parameter simply identifies the file. The second parameter is a flag that indicates what action you want to perform. The actions fall into these categories:

- obtain information about the data set. Refer to Table 22.4 for the macros used to gather this information.
- set information about data set type, label, and other attributes for an output data set. Refer to Table 22.5 for the macros used to set this information.

The last parameter defines the field where the information provided by SAS_XOINFO is stored. The type of the last field varies depending upon the macro used in the second parameter. Tables 22.3 an 22.4 indicate the type of the third parameter to use with each macro.

SAS_XOINFO *continued*

Table 22.3 *Codes for Data Set Information from the SAS_XOINFO Routine*

Flag	Storage	Contents
XO_ANY	`int*`	flag for empty file: −1 = no variables in file 0 = no observations in file 1 = file has observations and variables
XO_CRDTE	`double*`	creation date
XO_CUROBS	`ptr*`	address of the current record
XO_FOBS	`long*`	value of FIRSTOBS= SAS system option
XO_ISINDEX	`int*`	flag for index on file: 0 = no indexes are present 1 = at least one index is present
XO_LABEL	`char40`	data set label
XO_LIB	`char8`	libref
XO_LREC	`long*`	length of an observation
XO_LRID	`int*`	length of record ID
XO_MAXRC	`int*`	flag for automatic exit on I/O errors: 0 = automatically exit 1 = procedure checks for I/O errors and determines when to exit
XO_MEM	`char8`	member name
XO_MODDTE	`double*`	date last modified
XO_MODE	`int*`	open mode: XO_INPUT XO_OUTPUT
XO_MTYPE	`char8`	member type
XO_NDEL	`long*`	number of deleted records

(*continued*)

Table 22.3 (continued)

Flag	Storage	Contents
XO_NLOBS	`long*`	number of logical records. The number of records returned should be treated as a *maximum* number of observations, not as an exact number. If the data set is not subsetted by the FIRSTOBS= or OBS= SAS system options, this number indicates the number of observations and does not include those marked for deletion. If the data set is subsetted by either of these SAS system options, this number indicates the number of observations selected in the subset, but it also includes observations marked for deletion in the count. This number disregards any WHERE processing; that is, it is not possible to tell how many observations meet the WHERE clause. If this value is −1 the file is sequential.
XO_NVARS	`int*`	number of variables
XO_OBS	`long*`	value of OBS= option
XO_RADIX	`int*`	flag for support of access by observation number: 1 = access by observation is allowed; 0 = access by observation is not allowed
XO_TYPE	`char8`	TYPE= value (special data set type)

Table 22.4 indicates codes you can use with the SAS_XOINFO routine to set data set values.

Table 22.4 *Codes for Setting Information with the SAS_XOINFO Routine*

Flag	Storage	Action
XO_SLABEL	char40	sets the data set label (LABEL= value)
XO_STYPE	char8	sets the special data set type (TYPE= value)

Usage Notes

- Before you call SAS_XONOTE to fetch a record ID, call SAS_XOINFO using the XO_LRID code to obtain the length of the record ID. Then use one of the SAS_XM routines to allocate sufficient memory to store the record ID.
- After you call SAS_XVPUTE to complete the definition of the output file, you cannot modify the TYPE or LABEL fields in the output file. That is, do not call SAS_XOINFO with either the XO_STYPE or the XO_SLABEL macro after you call SAS_XVPUTE.

SAS_XOINFO *continued*

Example

This example illustrates calling SAS_XOINFO and casting the third argument to a `ptr`, the required type for this argument:

```
ptr fileid;
int nv;
.
SAS_XOINFO(fileid, XO_NVARS, (ptr) &nv);
```

In this example, SAS_XOINFO updates the variable `nv` with the number of variables in the file pointed to by `fileid`.

SAS_XONOTE

Return current record location

Declarations

Type	Variable	Use	Description
ptr	*file-ptr*	input	fileid of an open input data set
ptr	*record-id-ptr*	output	record ID
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc = SAS_XONOTE(file-ptr, record-id-ptr);
```

Description

The SAS_XONOTE routine allows you to determine the record ID of the observation you are currently processing. That is, it identifies the number of the last observation successfully read by SAS_XOGETN or SAS_XBYGET. You can then use this number in a later call to SAS_XOPNT.

The record ID is unique for each observation in the data set. The format for the record ID can differ from file to file. Therefore, you must call SAS_XOINFO with the action flag set to XO_LRID to obtain the size of the record ID and then allocate space to store each record ID you want to save before you call SAS_XONOTE.

Note: The SAS_XONOTE routine works differently than the SAS_XCNOTE routine. SAS_XONOTE notes the observation *in a SAS data set after* the observation is read. The SAS_XCNOTE routine notes the position of a record *in a SAS utility file before* the record is read.

Example

This example shows how to note the record ID of a record you have just read. You can then use the record ID to point back to the same observation in case you need to reread it later. For an illustration of how to point back to a noted observation, refer to the description of the SAS_XOPNT routine.

```
int     lrid;
ptr     infile,
        ptr_xv,
        rid;

SAS_XOINFO(infile, XO_LRID, (ptr) &lrid);
rid = (ptr) SAS_XMEMEX(lrid);

   /* read and write observations */
while (SAS_XBYNEXT(infile) > 0) {
   while (SAS_XBYGET(infile) != NULL) {
      SAS_XONOTE(infile, rid);
      SAS_XVGET(ptr_xv, NULL);
      process observation
   }
}
```

SAS_XON2RID

Convert observation number to record locator

Declarations

Type	Variable	Use	Description
ptr	*file-ptr*	input	fileid of an open input data set
long	*obs-num*	input	record number to be converted
ptr	*record-id-ptr*	output	record ID
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc = SAS_XON2RID(file-ptr, obs-num, record-id-ptr);
```

SAS_XON2RID *continued*

Description

The SAS_XON2RID routine allows you to determine the record ID of an observation when you know the observation number. You can then use this number in a later call to SAS_XOPNT.

The record ID is unique for each observation in the data set. The format for the record ID can differ from file to file. Therefore, you must call SAS_XOINFO with the action flag set to XO_LRID to obtain the size of the record ID and then allocate space to store the record ID before you call SAS_XNOTE.

Note: When you use this routine in a procedure, the data set processed by this routine must be radix addressable. Refer to Table 22.1 for more information on engines that provide this capability.

Example

This example illustrates how to obtain the record ID for a specific observation. The example finds the record ID of the first observation in the data set.

```
int      lrid,
         radix,
         rc;
ptr      infile,
         rid;
long     firstobs;

   /* test to be sure observation numbers can be changed */
   /* to record IDs for the input file being processed   */
SAS_XOINFO(infile, XO_RADIX, (ptr) &radix);
if (radix) {
   SAS_XOINFO(infile, XO_LRID, (ptr) &lrid);
   rid = (ptr) SAS_XMEMEX(lrid);

      /* obtain record ID of first observation */
   firstobs = SAS_XOPTIGT("FIRSTOBS", &rc);
   SAS_XON2RID(infile, firstobs, rid);
}
```

SAS_XOOPEN

Open a SAS data set

Declarations

Type	Variable	Use	Description
ptr	**ptr-to-file-ptr*	output	pointer to the fileid of the data set opened by SAS_XOOPEN
struct XOPNSTR	**ptr-to-xopnstr*	input	open parameters
rctype	*rc*	returned	return code: 0 = successful XHENOASN = the library is not assigned XHENOMBR = the member does not exist otherwise = unsuccessful

Usage

```
rc = SAS_XOOPEN(ptr-to-file-ptr, ptr-to-xopnstr);
```

Description

The SAS_XOOPEN routine allows dynamic opening of input or output mode data sets. Most procedures never call XOOPEN directly since semantic actions coded in the grammar cause the data set to be opened during the parsing process. However, if your procedure always opens the same data set and you want to open it automatically without having the user specify it, you can call SAS_XOOPEN directly.

Before your procedure can call SAS_XOOPEN, you must provide information about the file in the open file parameter structure (XOPNSTR). Refer to the description of the XOPNSTR in Chapter 20 for more information on the structure.

The required fields are as follows:

- `libname`
- `memname`
- `opnmode`
- `maxrc`

SAS_XOOPEN *continued*

If you are opening a data set in output mode, and you want to specify a data set type and label, complete the fields below. Note that you must complete the corresponding length field if you specify the type or label.

- type
- typelen
- label
- labellen

Example

This example illustrates opening a file from your procedure. The example tests the input data for errors. The first time errors are encountered, an output file is created to store the observations with errors.

```
struct XOPNSTR  openstr;
ptr     errfile;
rctype rc;
int     FIRSTERR;

read input data set and validate data

set FIRSTERR the first time an observation contains invalid data
if (FIRSTERR) {

      /* fill in required fields of XOPNSTR */
   SAS_ZSTRMOV("WORK", -1, openstr.libname, 8);
   SAS_ZSTRMOV("ERROR", -1, openstr.memname, 8);
   openstr.opnmode = XO_OUTPUT;
   openstr.maxrc = 0;

      /* open the file */
   rc = SAS_XOOPEN(&errfile, &openstr);
   SAS_XPRLOG(rc);
}
```

SAS_XOPNT

Position to a specific observation

Declarations

Type	Variable	Use	Description
ptr	*file-ptr*	input	fileid of an open input data set
ptr	*record-id*	input	address of record from XONOTE or one of the following record-ID flags to access a record: XO_BOD = first nondeleted record XO_LAST = the last nondeleted record XO_CURRENT = the last record read by SAS_XOGETN. XO_PREV = the first nondeleted record preceding the current record XO_NEXT = the first nondeleted record following the current record
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc = SAS_XOPNT(file-ptr, record-id);
```

Description

The SAS_XOPNT routine permits you to position the file pointer to a specific observation or a logical location in the data set. You can use SAS_XOPNT to position to an observation and then read it by using SAS_XOGETN or SAS_XBYGET. This function cannot be called for files opened for output.

This routine requires that you specify a record ID (or one of the record ID flags) to point to an observation. In general, you determine the record ID for a specific observation by calling SAS_XONOTE. If your application can run more efficiently by using actual observation numbers, call SAS_XOINFO with the XO_RADIX action flag. If it returns 1, the data set is *radix-addressable*, which means that observations can be located by their observation numbers. You still need to use the record ID when you call SAS_XOPNT, but you can obtain this by calling SAS_XON2RID to convert the observation number to the record ID.

To specify a logical location in the data set instead of a specific observation, use one of the following flags for the *record-id* parameter.

SAS_XOPNT *continued*

Flag	Location
XO_BOD	first nondeleted record
XO_LAST	last nondeleted record
XO_CURRENT	last record read by SAS_XOGETN
XO_PREV	first nondeleted record preceding the current record
XO_NEXT	first nondeleted record following current record

If the record you point to is marked for deletion, SAS_XOPNT returns a nonzero return code. You should check the return code because if you attempt to read the record, your procedure will fail.

If you are using the SAS_XBYNEXT and SAS_XBYGET routines on a data set, you should call the SAS_XOPNT routine only after passing through the entire BY group. In addition, you should not point to observations outside the current BY group. Refer to the following example for an illustration of how to call SAS_XOPNT when you are also doing BY processing.

Example

This example illustrates how to note the record ID of a record in a BY group and then return to that record after you finish processing the BY group.

```
ptr  fileid,
     p,
     rids;
int  ridlen,
     i;

SAS_XOINFO(fileid, XO_LRID, &ridlen);
rids = SAS_XMEMGET(ridlen * 100);
while (SAS_XBYNEXT(fileid) > 0) {
   i = 0;
   for (i = 0; SAS_XBYGET(fileid) != NULL && i < 100; i++) {
      process observation - do not point in this loop
      SAS_XONOTE(fileid, rids+i*ridlen);
   }

      /* position to second observation in BY group */
   SAS_XOPNT(fileid, rids+ridlen);

   p = SAS_XBYGET(fileid);
}
```

SAS_XORID2N

Convert record ID to observation number

Declarations

Type	Variable	Use	Description
ptr	*file-ptr*	input	fileid of an open input data set
ptr	*record-id*	input	record ID to be converted
long	**obs-number-ptr*	output	observation number
rctype	*rc*	returned	return code: 0 = successful; nonzero = unsuccessful

Usage

```
rc = SAS_XORID2N(file-ptr, record-id, obs-number-ptr);
```

Description

If you have the record ID for an observation and you want to obtain the observation number, use SAS_XORID2N. This routine is particularly helpful when you need to report the observation number to the user.

To request the observation number of the observation you are currently processing, use XO_CURRENT for the *record-id* parameter. For any other observation, you must first obtain the record ID of the observation by calling SAS_XONOTE, as illustrated here.

```
int  radix,
     lrid;
ptr  infile;

SAS_XOINFO(infile, XO_RADIX, (ptr) &radix);
if (radix) {
   SAS_XOINFO(infile, XO_LRID, (ptr) &lrid);
}
```

Then call SAS_XORID2N using the record ID returned by SAS_XONOTE to get the observation number.

Note: When you use this routine in a procedure, the data set processed by this routine must be radix addressable. Refer to Table 22.1 for more information on engines that provide this capability.

SAS_XORID2N *continued*

Example

This example illustrates how to convert the record ID of the record currently being processed to an observation number. Note the use of XO_CURRENT in the call to SAS_XORID2N.

```
int      radix;

long     obs_num;

ptr      infile,
         ptr_xv;

   /* test to be sure observation numbers can be changed */
   /* to record IDs for the input file being processed   */
SAS_XOINFO(infile, XO_RADIX, (ptr) &radix);

   /* read observations and print report with observation numbers */
while (SAS_XBYNEXT(infile) > 0) {
   while (SAS_XBYGET(infile) != NULL) {
      SAS_XVGET(ptr_xv, NULL);
      process input file
      if (radix) SAS_XORID2N(infile, XO_CURRENT, &obs_num);
      print output using observation number
   }
}
```

SAS_XOPT Routines: SAS System Options

The SAS_XOPT routines provide a means of determining which SAS system options are set when your procedure is invoked. For example, you can use the SAS_XOPT routines to test the value of the LINESIZE= and PAGESIZE= options before you format printed output.

To ensure that all OPTIONS statements are processed, do not call any SAS_XOPT routines until the parsing process is completed. That is, be sure all calls to SAS_XOPT routines follow the call to SAS_XSPARSE.

To test for a specific system option, you must use the appropriate SAS_XOPT routine and option number. The routine you call to test an option depends upon the type of the option. System options can be one of these types:

- bit options. These system options are ones the user specifies in an *option*/NO*option* format. For example, the CENTER/NOCENTER option is a bit option. Use the SAS_XOPTBGT routine to test the value of bit options.
- character options. These options require that the user specify a character value for the option. For example, the FORMCHAR= or MISSING= options are character options. Use the SAS_XOPTCGT routine to test the value of character options.

□ integer options. For these options, the user specifies a numeric value. For example, FIRSTOBS= and LINESIZE= are integer options. Use the SAS_XOPTIGT routine to test the value of integer options.

Table 22.5 lists the system options you can use with the SAS_XOPT routines and indicates the routine to use to test the value of the option. Note that host-dependent options cannot be specified.

Table 22.5 *Appropriate SAS_XOPT Routines for SAS System Options*

SAS System Option	SAS_XOPTBGT	SAS_XOPTCGT	SAS_XOPTIGT
BATCH	X		
BUFNO=			X
BUFSIZE=			X
BYLINE	X		
CAPS	X		
CARDIMAGE	X		
CATCACHE=			X
CENTER	X		
CHARCODE	X		
CLEANUP	X		
CMDMAC	X		
COMPRESS		X	
CPUID	X		
DBCS	X		
DBSCLANG=		X	
DBCSTYPE=		X	
DEVICE=		X	
DKRICOND=		X	
DKROCOND=		X	
DMR	X		
DMS	X		
DMSBATCH	X		
DSNFERR	X		
ECHOAUTO	X		
ENGINE=		X	

(continued)

Table 22.5 *(continued)*

SAS System Option	SAS_XOPTBGT	SAS_XOPTCGT	SAS_XOPTIGT
ERRORABEND	X		
ERRORS=			X
FIRSTOBS=			X
FMTERR	X		
FMTSEARCH=		X	
FORMCHAR=		X	
FORMDLIM=		X	
FORMS=		X	
FS	X		
GRAPH	X		
GWINDOW	X		
IMPLMAC	X		
INITSTMT=		X	
INVALIDDATA=		X	
LABEL	X		
LAST=		X	
LINESIZE=			X
LOCKWAIT=			X
MACRO	X		
MAPS=		X	
MAUTOSOURCE	X		
MERROR	X		
MISSING=		X	
MLOGIC	X		
MPRINT	X		
MRECALL	X		
NEWS=		X	
MSGCASE	X		
MSGLEVEL=		X	
MSTORED	X		

(continued)

Table 22.5 *(continued)*

SAS System Option	SAS_XOPTBGT	SAS_XOPTCGT	SAS_XOPTIGT
MSYMTABMAX=			X
MVARSIZE=			X
NOTES	X		
NUMBER	X		
OBS=			X
OVP	X		
PAGENO=			X
PAGESIZE=			X
PARM=		X	
PARMCARDS=		X	
PMENUS	X		
PROBSIG=			X
PROC	X		
REMOTE=		X	
REPLACE	X		
RSASUSER	X		
REUSE=		X	
S=			X
SASAUTOS=		X	
SASFRSCR=		X	
SASHELP=		X	
SASMSG=		X	
SASMSTORE=		X	
SASSCRIPT=		X	
SASUSER=		X	
SCROLLBARS	X		
SEQ=			X
SERROR	X		
SETINIT	X		

(continued)

Table 22.5 *(continued)*

SAS System Option	**SAS_XOPTBGT**	**SAS_XOPTCGT**	**SAS_XOPTIGT**
SITEINFO=		X	
SKIP=			X
SORTSEQ=		X	
SORTSIZE			X
SOURCE	X		
SOURCE2	X		
SPOOL	X		
SYMBOLGEN	X		
SYSPARM=		X	
S2=			X
TAPECLOSE=		X	
TERMINAL	X		
TRANTAB=		X	
USER=		X	
VNFERR	X		
WORK=		X	
WORKINIT	X		
WORKTERM	X		
YEARCUTOFF=			X

SAS_XOPTBGT

Fetch the value of a bit option

Declarations

Type	Variable	Use	Description
char	**name-ptr*	input	pointer to the name of the option to be fetched. You must provide the name of a valid SAS system option and the name must be null terminated.
int	**return-code-ptr*	output	pointer to the return code from this routine: 0 = option found and returned −1 = invalid name or no such option −2 = option is not a bit option. Check the type of the option and then use either SAS_XOPTCGT or SAS_XOPTIGT.
int	*value*	returned	value of the option.

Usage

```
value = SAS_XOPTBGT(name, return-code-ptr);
```

Description

The SAS_XOPTBGT routine allows you to test on/off SAS system options that the user can specify in the OPTIONS statement or at SAS invocation.

Note: This routine differs from many SAS_X routines because the return code for the function is stored in one of the parameters of the function. Keep in mind that the value of the option is returned in *i*, the function's return value and the return code is stored in the integer pointed to by *return-code*, one of the parameters. You must allocate space for the integer where the return code is stored and set the *return-code* to point to this integer before calling SAS_XOPTCGT.

Example

This example illustrates how to check the LABEL SAS system option to determine if labels should be used in printing output.

```
int   uselabel,
      rc;

   /* check to see if labels can be printed */
uselabel = SAS_XOPTBGT("LABEL", &rc);
if (rc < 0)
   SAS_XEBUG("Option name or routine is invalid.");
```

SAS_XOPTCGT

Fetch the value of a character option

Declarations

Type	Variable	Use	Description
char	**name-ptr*	input	pointer to the name of the option to be fetched. You must provide the name of a valid SAS system option and the name must be null terminated.
int	**return-code-ptr*	output	pointer to the return code from this routine: 0 = option found and returned −1 = invalid name or no such option −2 = option is not a character option. Check the type of the option and then use either SAS_XOPTBGT or SAS_XOPTIGT.
char	**char-value-ptr*	returned	pointer to the option's value. The value is null terminated.

Usage

```
char-value-ptr = SAS_XOPTCGT(name-ptr, return-code-ptr);
```

Description

The SAS_XOPTCGT routine allows you to test SAS system options that are set to a character string.

Note: This routine differs from many SAS_X routines because the return code for the function is stored in one of the parameters of the function. Keep in mind that the value of the option is returned in *char-value*, the function's return value, and the return code is stored in the integer pointed to by *return-code*, one of the parameters. You must allocate space for the integer where the return code is stored and set the *return-code* to point to this integer before calling SAS_XOPTCGT.

▶ *Caution* .... ***The value stored at the address pointed to by char-value may change, so if you need to test this value later in your program, store a copy of this value in your program.*** ▲

Example

This example illustrates how to text the FORMCHAR= SAS system option to determine what formatting characters to use when printing output with outlining characters.

```
int         rc;
char        *formchar;

   /* get outlining characters */
formchar = SAS_XOPTCGT("FORMCHAR", &rc);
if (rc < 0)
   SAS_XEBUG("Option name or routine is invalid.");
```

SAS_XOPTIGT

Fetch the value of an integer option

Declarations

Type	Variable	Use	Description
char	**name-ptr*	input	pointer to the name of the option to be fetched. You must provide the name of a valid system option and the name must be null terminated.
int	**return-code-ptr*	output	pointer to the return code from this routine: 0 = option found and returned −1 = invalid name or no such option −2 = option is not an integer option. Check the type of the option and then use either SAS_XOPTBGT or SAS_XOPTCGT.
long	*value*	returned	value of the option.

Usage

```
value = SAS_XOPTIGT(name-ptr, return-code-ptr);
```

Description

The SAS_XOPTIGT routine allows you to test SAS system options that are set to a numeric value.

Note: This routine differs from many SAS_X routines because the return code for the function is stored in one of the parameters of the function. Keep in mind that the value of the option is returned in *value*, the function's return value, and the return code is stored in the integer pointed to by *return-code*, one of the parameters. You must allocate space for the integer where the return code is stored and set the *return-code* to point to this integer before calling SAS_XOPTIGT.

SAS_XOPTIGT *continued*

Example

This example illustrates how to test the PAGESIZE= SAS system option to determine how many lines to print on each page of output.

Note: The SAS_XPOPTGT routine provides other methods of testing the size of a printed page.

```
int        rc,
           pagesize;

   /* get number of lines on a page */
pagesize = SAS_XOPTIGT("PAGESIZE", &rc);
if (rc < 0)
   SAS_XEBUG("Option name or routine is invalid.");
```

SAS_XP Routines: Printing

The SAS_XP routines allow you to print output to the SAS procedure output file or the SAS log. Your procedure should use the SAS_XP routines to print all output. Although you can use native language printing routines for generating printed output from procedures, it is better to use the SAS_XP routines because they provide the following capabilities:

- special formatting of missing values
- special format codes, which indent continuation lines and add ERROR, WARNING, or NOTE labels
- highlighting or coloring of lines such as warnings and errors in the log or titles and column headings in procedure output
- pagination
- automatic handling of titles, notes, and footnotes
- output to the standard SAS procedure output file, which means the output from your procedure can be printed, sent to a file, displayed by the SAS display manager systems and treated as any other SAS output.

There are two groups of SAS_XP routines. The primary group can be used in any language, although you may need to add a special step to use these routines with the PL/I language if you are running your procedure under the VMS operating system. You can use another group of routines if you are writing your procedure in PL/I. This group is particularly useful if you have a Version 5 procedure in PL/I that you are converting to Version 6.

The SAS_XP routines that can be used with any language are as follows:

- SAS_XPAGE
- SAS_XPHDR
- SAS_XPOPTGT
- SAS_XPOPTST
- SAS_XPPRN
- SAS_XPRLOG
- SAS_XPS
- SAS_XPSKIP
- SAS_XPSLOG
- SAS_XPSPRN
- SAS_XPSSTR.

These routines are discussed in detail in the next section. Note that these routines are referred to throughout this chapter as the standard SAS_XP routines.

The SAS_XP routines that can be used only with the PL/I language are as follows:

- SAS_XPC
- SAS_XPCS
- SAS_XPF
- SAS_XPFS
- SAS_XPF12
- SAS_XPI
- SAS_XPIS
- SAS_XPI12
- SAS_XPSLOG1 through SAS_XPSLOG10.

These routines are discussed in detail later in this chapter in "Printing Routines for the PL/I Language." This section also discusses special requirements for using the routines available for any language with PL/I.

Standard SAS_XP Routines

In most cases, you use any of the following combinations of SAS_XP routines to print output:

- Call SAS_XPSPRN to format and print output in one step.
- Call SAS_XPS to format output in the output buffer. Then call SAS_XPPRN, SAS_XSKIP, or SAS_XPAGE to print the output.
- Call SAS_XPSSTR to format a character string. Then use SAS_XPS to place the string in the output buffer, which you can print with SAS_XPSPRN. Note that you can also use SAS_XPSSTR to format a character string that you do not want to print.
- Call SAS_XPSLOG to print messages to the log.
- Call SAS_XPSSTR to format a character string to be used for a heading, such as column headings. Then call SAS_XPHDR to define that heading so it is printed on each page of procedure output.

The arguments to these routines are based on the C-language technique of formatting output for printing. That is, the string to be printed is followed by any number of arguments that are used as substitution values in the string. The string consists of literals and formatting codes indicated by %. In particular, SAS_XPS, SAS_XPSLOG, SAS_XPSPRN, and SAS_XPSSTR use these string formatting codes. The codes shown in Table 22.6 are used for character substitution. Table 22.7 lists codes that can be used to affect an entire line of text or that format a line in a special manner.

Table 22.6 *C-like Substitution Characters for SAS_XP Routines*

Substitution Character	Other Forms ♦	Standard C Code	Data Types	Description
%*a	%*n*a	no	C: do not use PL/I: ADDR FORTRAN: INT*4	passes an address of a string. The length of the string must be specified, either in the format code itself or preceding the string as a separate argument. For FORTRAN programs, you can obtain the address using the IADDR function.
%b	%*b %*n*b	no	C: null-terminated string PL/I: CHAR FORTRAN: CHAR	formats a character string that ends in a blank or a null terminator. If no length is specified, the first blank or null terminator encountered indicates the end of the string. If a length is specified, the first blank or null terminator after the specified length indicates the end of the string.
%c		yes	C: `int` or single `char` PL/I: CHAR FORTRAN: CHAR	formats a single character. Do not use this type with a string.
%d	%*d %*n*d %−d %−*n*d	yes	C: `int` or `short` PL/I: FIXED BIN(31) or (15) FORTRAN: INT*4 or INT*2	formats an integer to the default data type**.
	%ld %*ld %*n*ld %−ld %−*n*ld	no	C: `long` PL/I: FIXED BIN(31) FORTRAN: INT*4	formats a long integer.

(continued)

♦ In general, other forms of a formatting code use the following conventions:

%* indicates the field width is specified as a separate argument following the value to be substituted.

%*n* indicates the field width is specified as part of the formatting code, such as `10d` to format an integer in a field width of 10.

%− indicates the value is right-aligned.

Table 22.6 *(continued)*

Substitution Character	**Other Forms** ♦	**Standard C Code**	**Data Types**	**Description**
%f	%*f %*w.d*f %−f %−*w.d*f %*.$f	yes	C: `double` PL/I: FLOAT BIN(53) FORTRAN: REAL*8	formats a floating-point number. In the alternate forms, the asterisk can be replaced by a number in the form *w.d* to indicate the width and number of decimal places. Likewise, the `f` formatting code can be prefixed with a *w.d* width and decimal specification.
%*.$f		no	C: `double` PL/I: FLOAT BIN(53) FORTRAN: REAL*8	formats a floating-point number with a format code. In this form, the dollar sign ($) works as a substitution character for the format code. For this form, you use three arguments: the width, a `long` variable that contains the format code obtained from calling SAS_XFFILE, and finally the value to be formatted.
%s	%*s %*il.ol*s	yes	C: `char *` PL/I: CHAR FORTRAN: CHAR	formats a string. If the string is null-terminated, do not use the *il* formatting code (or specify −1 for this code). The *il* formatting code indicates the length of the input string. The *ol* formatting code indicates the width of the output field into which the string is formatted.
%x	%*x %*n*x %−x %−*n*x	yes	C: `long` or `short` PL/I: FIXED BIN(31) or (15) FORTRAN: INT*4 or INT*2	formats the default integer type to a hexadecimal number. ♦♦
	%lx %*lx %*n*lx %−lx %−*n*lx		C: `long` PL/I: FIXED BIN(31) FORTRAN: INT*4	formats a long integer as a hexadecimal number.

♦ In general, other forms of a formatting code use the following conventions:

%* indicates the field width is specified as a separate argument following the value to be substituted.

%*n* indicates the field width is specified as part of the formatting code, such as `10d` to format an integer in a field width of 10.

%− indicates the value is right-aligned.

♦♦ On most host operating systems, an `int` is a `long`. On a few operating systems, an `int` is a `short`.

Table 22.7 lists formatting codes that change how a line is printed.

Table 22.7 *C-like Line Formatting Codes for SAS_XP Routines*

Substitution Character	**Other Forms**	**Standard C Code**	**Description**
%*n*h		no	highlights an entire line using the color associated with specific parts of the output. Color codes are as follows: %0h use the color of the data lines %1h use the color of the header lines %2h use the color of the source lines %3h use the color of the title lines %4h use the color of the BY lines %5h use the color of the footnote lines %6h use the color of errors %7h use the color of warnings %8h use the color of notes
%m		no	marks a position to which subsequent wrapped lines should be indented.
%n		similar to \n	advances to a new line and cancels the effect of %m.
%−2n		no	advances to a new line without cancelling the effect of %m.
%*n*t	%+*n*t %−*n*t	similar to \t	formats text to begin in the *n* column. When you specify a minus or plus sign, the text begins *n* columns before or after the current column.
%1z		no	prefixes messages with ERROR:
%2z		no	prefixes messages with WARNING:
%3z		no	prefixes messages with NOTE:
%%		yes	enables you to print a single % (percent sign).

Printing Routines for the PL/I Language

This section discusses how to use the SAS_XP routines already discussed with the PL/I language and then describes some special SAS_XP routines that you can use with PL/I.

Using Standard SAS_XP Routines with the PL/I Language

You can use any of the routines discussed in the previous section with PL/I but you must modify your use of them if you are running your procedure under the VMS operating system or if you may ever need to port your procedure to the VMS operating system. It is recommended that even if you are not currently running under VMS, you should write your procedure so that it would successfully work under all operating systems.

PL/I programmers working under the VMS operating system encounter problems with the SAS_XP routines discussed earlier because the SAS_XP routines expect to receive data of specific types and VMS may pass data of different types. To ensure that your printing routines work correctly, you must use a PL/I function that describes each argument after the initial character string.

For example, SAS/TOOLKIT software uses the descriptive information contained in the PL/I dope vector for character arguments to determine the length of the string. On VMS, the only way to ensure that the arguments to the standard SAS_XP routines have the correct descriptor information is to use the DESCRIPTOR function, as shown here.

```
CALL SAS_XPSLOG2('Value is %s.',DESCRIPTOR('XYZ'));
```

The DESCRIPTOR function is supported on all systems that SAS/TOOLKIT software runs on, so it is recommended that you use this function in PL/I procedures that call any standard SAS_XP routines.

In the calls to the SAS_XP routines, you must use a numeric variable for numeric arguments, not a numeric literal to ensure that the type of the numeric argument matches the type required by the routine. For example, the following call to SAS_XSPLOG is incorrect:

```
CALL SAS_XPSLOG2("Value is %d.",4);    /* incorrect call */
```

Instead, you must use one of the following sets of code to ensure that the numeric value passed to SAS_XPSLOG is a FIXED BIN(31).

Using a Constant

```
CALL SAS_XPSLOG2("Value is %d.",
                 FIXED(4,31));
```

Using a Variable

```
DCL I FIXED BIN(31);

I = 4;
CALL SAS_XPSLOG2("Value is %d.",I);
```

Note that floating-point numbers have the same problem. Do not specify them as literals. Use a variable or the FLOAT function.

Specifying Varying Numbers of Arguments with the PL/I Language

Some operating systems, such as VMS, require that you specify exactly how many arguments must be used in a call to a PL/I routine. Several of the SAS_XP routines permit varying numbers of arguments: SAS_XPS, SAS_XPSLOG, SAS_XPSPRN, and SAS_XPSSTR. To use these routines with PL/I, use one of the corresponding numbered routines. The numbers suffixed to the routine name indicate the total number of arguments specified in the call.

Note: These routines work under all operating systems and are required under VMS.

Special Printing Routines for PL/I Programs

If you are writing your program in PL/I, you may prefer to use the routines listed in Table 22.8 to print output. These routines are more adapted to PL/I printing techniques and do not require the DESCRIPTOR function. In addition, these routines provide printing techniques that are comparable to Version 5 PL/I procedures. You still must convert your Version 5 procedures to work with Release 6.06 and later; for example, you must use the substitution characters listed in Table 22.6 instead of the Version 5 substitution characters. However, the following routines permit you to maintain the same calling structure used in the corresponding Version 5 routine, which requires less conversion time.

Table 22.8 *Special PL/I Printing Routines*

Version 6 Routine	Version 5 Routine
SAS_XPAGE	XPAGE
SAS_XPC	XPC
SAS_XPCS	XPCS
SAS_XPF	XPF
SAS_XPFS	XPFS
SAS_XPF12	XPF12
SAS_XPI	XPI
SAS_XPIS	XPIS
SAS_XPI12	XPI12
SAS_XPSKIP	XPSKIP

These routines correspond approximately to PL/I PUT statements as shown in Table 22.9.

Table 22.9 *Comparison to PL/I Printing Commands*

SAS Routine	Corresponding PL/I Statement
`SAS_XPC(message,length,position)`	`PUT EDIT(message) (COL(position),A(length))`
`SAS_XPF(value,length,number,position)`	`PUT EDIT(value) (COL(position),F(length,number))`
`SAS_XPI(value,length,position)`	`PUT EDIT(value) (COL(position),F(length))`
`SAS_XPSKIP(number)`	`PUT SKIP(number)`
`SAS_XPAGE`	`PUT PAGE`

SAS_XPAGE

Finish the current page and prepare to go to top of the next page

Usage

C	`SAS_XPAGE();`
FORTRAN	`CALL SAS_XPAGE()`
PL/I	`CALL SAS_XPAGE();`

Description

Use the SAS_XPAGE routine to start a new page. SAS_XPAGE ensures that no blank or empty (just titles and footnotes) pages are printed, even if you call it twice in a row.

You should always call SAS_XPAGE after you call SAS_XSPARSE to ensure that your procedure recognizes any options that affect printed output that were set with an OPTIONS statement included in the statements used to call your procedure. The first call to SAS_XPAGE must precede any formatting of output to ensure that the options are correctly processed. In addition, you should call SAS_XPAGE at the beginning of each BY group to handle BY group titles.

When you call SAS_XPAGE, it does the following:

- prints the output currently in the output buffer
- checks the LINESIZE=, PAGESIZE=, CENTER, and NUMBER SAS system options
- sets a flag to indicate that titles and other heading lines should be printed before the next line of output is printed.

Example

```
rctype rc;
long   left;
short  nvar;

left = SAS_XPOPTGT(XPLEFT, &rc);
if (left < nvar) SAS_XPAGE();
```

SAS_XPC

Store character string in the output buffer

PL/I language only

Declarations

Type	Variable	Use	Description
CHAR(*n*)	*message*	input	string to move to output line
FXL	*length*	input	length of string
FXL	*position*	input	starting point in output line: > 0 = explicit column position 0 = current column position < 0 = offset from current column

Usage

PL/I `CALL SAS_XPC(`*`message-ptr, length, position`*`);`

Description

The SAS_XPC routine places a character string in the output buffer. You can specify the location in the output buffer where you want to place the string, or it simply can be added to the end of the other strings placed in the output buffer.

If you specify a *length* greater than 0, the text is placed in that amount of space. If the field is longer than the text, the text is left justified in the field and padded with blanks on the right. If *length* is less than the length of the string, the string is truncated on the right. If you specify a 0 or a negative number for the *length,* the routine uses the length of *message* for the length.

If you specify a *position* where a value has already been stored, the value is replaced with this text. If you specify 0, the text is placed in the current position in the output buffer. If you want to move one or more spaces after the last value in the output buffer, specify a negative number with a magnitude of the number of spaces you want to move. For example, to move four spaces, specify -4 for *position*. This skips three spaces and places the output in the fourth position after the last text in the output buffer.

SAS_XPC *continued*

Example

This example prints a text string to the current position in the output buffer. Note that the string is right justified in a field width of 20.

```
DCL      DESC     CHAR(13);

DESC='A SAMPLE LINE';

CALL SAS_XPC(ADDR(DESC),20,0);
```

SAS_XPCS

Format character string into another string

PL/I language only

Declarations

Type	Variable	Use	Description
CHAR(*n*)	*message*	input	string to format
FXL	*length*	input	field width
FXL	*code*	input	format code
CHAR(*n*)	*out-string*	output	output string

Usage

PL/I `CALL SAS_XPCS(message-ptr, length, code, out-string-ptr);`

Description

The SAS_XPCS routine formats a string and places it in another string. SAS_XPCS is useful for creating formatted lines to write to the log. After formatting a string with SAS_XPCS, use SAS_XPSLOG to output the line to the log.

You need to call SAS_XFNAME to obtain the format code for character formats other than $CHARw. To format the string using $CHARw, specify 0 for the *code*.

Example

This example formats a character string using the $ASCII format and stores the results in another character string.

```
DCL    LETTERS    CHAR(6);
DCL    ASCLETS    CHAR(12);
DCL    RC         FXL;
DCL    FORMCODE   FXL;

LETTERS = 'ABCabc';
RC = SAS_XFNAME('$ASCII  ', 'F', SAS_NUL(), ADDR(FORMCODE));
CALL SAS_XPCS(ADDR(LETTERS), 20, FORMCODE, ADDR(ASCLETS));
```

SAS_XPF

Format a double into the output buffer

PL/I language only

Declarations

Type	Variable	Use	Description
FTL	*value*	input	value to format.
FXL	*length*	input	format width and justification. The maximum width is 40: >0 = right justify data in field <0 = left justify data in field and remove trailing blanks.
FXL	*number*	input	indicates how to format the value. The value of this field can indicate a specific SAS format, the number of decimal places, or other formatting information: <0 = exponential format (Ew.) using the absolute value of *number* for *w*. 0-40 = *w.d* format using *number* as *d*. 98 = BEST format with two decimal places. This format makes it easier to align numbers in columns. 99 = BEST*w*. format. >100 = any value over 100 is treated as a format code, which is obtained from SAS_XFNAME. To specify decimal places for a format, add the number of decimal places to the format code returned by SAS_XFNAME.

SAS_XPF *continued*

FXL	*position*	input	starting point in output line: > 0 = explicit column position 0 = current column position < 0 = offset from current column.

Usage

PL/I `CALL SAS_XPF(value, length, number, position);`

Description

The SAS_XPF routine formats a floating-point value into the output buffer. You can specify the location in the output buffer where you want to place the value, or it simply can be added to the end of the other strings placed in the output buffer.

If you specify a *length* greater than 0, the text is placed in that amount of space. If you specify a *length* less than 0, trailing blanks are removed, and the output is right justified. If the formatted value does not fit into the width specified, or the width is invalid, the field is filled with * to signal that the data could not be accurately presented.

Note that the *number* argument is used in several ways. Refer to the information in "Declarations" for details on how to use it.

If you specify a *position* where a value has already been stored, the value is replaced with this text. If you specify 0, the text is placed in the next space after the last value in the output buffer. If you want to skip spaces after the last value in the output buffer, specify a negative number with a magnitude of the number of spaces you want to skip. For example, to skip 4 spaces, specify -4 for *position*.

Example

This example prints a numeric value to the current position in the output buffer. Note that the value is left justified in a field width of 10 and formatted with two decimal places in the BEST format (indicated by the value of 98 for the *number* parameter).

```
CALL SAS_XPF(19.42, -10, 98, 0);
```

SAS_XPFS

Format a double into the specified string

PL/I language only

Declarations

Type	Variable	Use	Description
FTL	*value*	input	value to format. Maximum width is 40.
FXL	*length*	input	format width and justification: > 0 = right justify data in field < 0 = left justify data in field and remove trailing blanks.
FXL	*number*	input	indicates how to format the value. The value of this field can indicate a specific SAS format, the number of decimal places, or other formatting information: < 0 = exponential format (Ew.) using the absolute value of *number* for *w*. 0-40 = *w.d* format using *number* as *d*. 98 = BEST format with two decimal places. This format makes it easier to align numbers in columns. 99 = BEST*w*. format. > 100 = any value over 100 is treated as a format code, which is obtained from SAS_XFNAME. To specify decimal places for a format, add the number of decimal places to the format code returned by SAS_XFNAME.
CHAR(*n*)	*out-string*	output	output string

Usage

PL/I `CALL SAS_XPFS(value, length, number, out-string-ptr);`

Description

The SAS_XPFS routine formats a floating-point value into a character string. SAS_XPFS is useful for creating formatted lines to write to the log or for converting floating-point values for routines that require character arguments.

If you specify a *length* greater than 0, the text is placed in that amount of space. If the field is longer than the text, the text is left justified in the field and padded with blanks on the right. If you specify a *length* less than 0, trailing blanks are removed and the output is right justified. If the formatted value does not fit

SAS_XPFS *continued*

into the width specified, or the width is invalid, the field is filled with * to signal that the data could not be accurately presented.

Note that the *number* argument is used in several ways. Refer to the information in "Declarations" for details on how to use it.

Example

This example formats a numeric value to a string using the DOLLAR SAS format. Note that the value is right justified in a width of 20 and it has two decimal places.

```
DCL     FORMCODE   FXL;
DCL     RC         FXL;
DCL     DOLLARS    CHAR(30);

RC = SAS_XFNAME('DOLLAR  ', 'F', SAS_NULL(), ADDR(FORMCODE));
CALL SAS_XPFS(18946, 20, FORMCODE+2, ADDR(DOLLARS));
```

SAS_XPHDR

Create header lines to be printed on each page

Declarations

Type	Variable	Use	Description
int	*header-num*	input	The relative position of this heading in the 10 possible headings. Heading 1 appears at the top, heading 10 at the bottom: 0 = clear all headings.
char	**string-ptr*	input	pointer to the string to be used for the heading: NULL = clear the headings with number *header-num* and greater.
int	*length*	input	length of the input string or 0 when *string-ptr* is NULL.
int	*rc*	returned	return code.

Usage

```
rc = SAS_XPHDR(header-num, string-ptr, length);
```

Description

The SAS_XPHDR routine enables you to define headings that are automatically printed on each page of output in the procedure output file. You call SAS_XPHDR for each line of heading, up to 10 headings. To create the heading, you can either declare a static character string or use the SAS_XPSSTR routine to format a string. The headings you define with SAS_XPHDR are most useful for printing column headings or for defining portions of the title that are not changed when the user specifies a TITLE statement. These headings follow any titles that are defined by the SAS TITLE statement.

When you define headings with SAS_XPHDR, the SAS System automatically adds one blank line after the last heading you defined. You can call SAS_XPHDR to change the headings from page to page. Simply call SAS_XPHDR before the SAS_XPAGE call that starts the new page.

You can alter the default behavior of SAS_XPHDR by using the XPBYLINPOS and XPNOBLANK flags with the SAS_XPOPTST routine described later in this chapter.

Example

This example formats a header using the SAS_XPSSTR routine and then calls SAS_XPHDR to establish the header so it will print on each new page.

```
char header[256];
int  length,
     offset,        /* amount to indent to center output */
     SUM;           /* flags if extra variable should be printed */

SAS_XPSSTR(header, "%*tOBS %*tVARIABLE %*tPREVIOUS VALUE", (3+offset),
           (10+offset), (22+offset));
SAS_XPSSTR(header, "%*tNEW VALUE", (45+offset));
length = 54+offset;

if (SUM) {
   SAS_XPSSTR(header, "%*tSUM", (69+offset));
   length = 72+offset;
}
SAS_XPHDR(1, header, length);
```

SAS_XPI

Format an integer into the output buffer

PL/I language only

Declarations

Type	Variable	Use	Description
FXL	*value*	input	integer to format.
FXL	*length*	input	field width and justification. The maximum field width is 15: > 0 = right justify data in field < 0 = left justify data and remove trailing blanks.
FXL	*position*	input	starting point in output line: > 0 = explicit column position 0 = current column position < 0 = offset from current column.

Usage

PL/I `CALL SAS_XPI(`*`value, length, position`*`);`

Description

The SAS_XPI routine places a formatted integer in the output buffer. You can specify the location in the output buffer where you want to place the value, or it simply can be added to the end of the other strings placed in the output buffer.

If you specify a *length* greater than 0, the text is placed in that amount of space. If the field is longer than the text, the text is left justified in the field and padded with blanks on the right. If you specify a *length* less than 0, trailing blanks are removed, and the output is right justified. If the formatted value does not fit into the width specified, or the width is invalid, the field is filled with * to signal that the data could not be accurately presented.

If you specify a *position* where a value has already been stored, the value is replaced with this text. If you specify 0, the text is placed in the next space after the last value in the output buffer. If you want to skip spaces after the last value in the output buffer, specify a negative number with a magnitude of the number of spaces you want to skip. For example, to skip 4 spaces, specify −4 for *position.*

Example

This example prints an integer to the current position in the output buffer. Note that the value is right justified in a field width of 10.

```
CALL SAS_XPI(1565, 10, 0);
```

SAS_XPIS

Format an integer into a string

PL/I language only

Declarations

Type	Variable	Use	Description
FXL	*value*	input	integer to format.
FXL	*length*	input	field width. The maximum field width is 15: > 0 = right justify data in field < 0 = left justify data and remove trailing blanks.
CHAR(*n*)	*out-string*	output	target string.

Usage

PL/I `CALL SAS_XPIS(`*`value, length, out-string-ptr`*`);`

Description

The SAS_XPIS routine formats an integer into a character string. Use SAS_XPIS to create formatted lines that you can write to the log or to convert integer values for routines that require character arguments.

If you specify a *length* greater than 0, the text is placed in that amount of space. If the field is longer than the text, the text is left justified in the field and padded with blanks on the right. If you specify a *length* less than 0, trailing blanks are removed, and the output is right justified. If the formatted value does not fit into the width specified, or the width is invalid, the field is filled with * to signal that the data could not be accurately presented.

Example

```
DCL     OUTSTR  CHAR(20);

CALL SAS_XPIS(1999,-8,ADDR(OUTSTR));
```

SAS_XPOPTGT

Obtain information about output options and buffer

Declarations

Type	Variable	Use	Description
int	*mnemonic*	input	a mnemonic for the option or buffer setting you want to test. Valid values follow; these values are explained in "Description": XPBYLINPOS XPLMARGIN XPCENTER XPNOBLANK XPCINDENT XPNODATE XPCOLNUM XPNOFOOTS XPLEFT XPNOPAGE XPLINENUM XPNOTITLE XPLINEONE XPPAGENUM XPLINESIZE XPPAGESIZE
int	**rc-ptr*	output	pointer to a return code: 0 = succesful nonzero = unsuccessful.
int	*information*	returned	a flag indicating the bit option is set, the value of a parameter option, or information about the contents of the output buffer.

Usage

```
information = SAS_XPOPTGT(mnemonic, rc-ptr);
```

Description

Use this routine to get information about the output buffer and the SAS system options that control printing. Specify one of the mnemonics listed in this section to indicate what information you want to get.

Note: Be sure that you call SAS_XPAGE before you use SAS_XPOPTGT. Calling SAS_XPAGE sets the values of some of the SAS system options that are tested by SAS_XPOPTGT.

XPBYLINPOS
: the number of heading lines to print before the BY line.

XPCENTER
: a flag indicating whether or not the CENTER SAS system option is on.

XPCINDENT
: the amount to indent continuation lines when lines must be split because they are too long.

XPCOLNUM	the column number where the next string of output will be placed.
XPLEFT	the number of lines left to print on the page.
XPLINENUM	the line number of the line currently being created.
XPLINEONE	a flag that indicates a new page should be started.
XPLINESIZE	the value of the LINESIZE= SAS system option. This is the maximum number of bytes in the output buffer that your procedure can use. Each time you add a text string to the output buffer, your procedure needs to compare the value returned for XPCOLNUM to the line size to ensure that you have sufficient room in the buffer to move the value. Note that if your output is longer than the line size, the text string is not wrapped to another line.
XPLMARGIN	the starting column in the output buffer where a string of output is placed.
XPNOBLANK	suppresses the blank line following the header lines.
XPNODATE	suppresses printing the date.
XPNOFOOTS	suppresses printing footnotes.
XPNOPAGE	suppresses automatic page breaks. When this flag is set, you must use SAS_XPAGE to force page breaks.
XPNOTITLE	suppresses printing titles.
XPPAGENUM	returns the page number currently being printed.
XPPAGESIZE	returns the value of the PAGESIZE= SAS system option. You can use the value returned for XPLEFT to determine how many lines are left on the page.

Example

This example tests the CENTER SAS system option to determine if the output from the procedure should be centered.

```
int  CENTER,
     rc;

CENTER = SAS_XPOPTGT(XPCENTER, &rc);
if (CENTER) {
   calculate the indention needed to center output
}
```

SAS_XPOPTST

Set the value of an option

Declarations

Type	Variable	Use	Description
int	*mnemonic*	input	mnemonic for option or buffer setting you want to change: XPBYLINPOS XPNODATE XPCINDENT XPNOFOOTS XPCOLNUM XPNOPAGE XPLMARGIN XPNOTITLE XPNOBLANK
int	*setting*	input	setting for the option or buffer control value
int	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc = SAS_XPOPTST(mnemonic, setting);
```

Description

Use this routine to change settings that affect the output buffer and the printing options. Specify one of the mnemonics listed in this section to indicate what information you want to set.

Note: After you set one of the values listed here, you must call SAS_XPAGE to put that new setting into effect.

XPBYLINPOS	the number of heading lines to print before the BY line.
XPCINDENT	the amount to indent continuation lines when lines must be split because they are too long.
XPCOLNUM	the column number where the next string of output will be placed.
XPLMARGIN	the starting column in the output buffer where a string of output is placed.
XPNOBLANK	a flag that suppresses the blank line following the header lines. For options prefixed with XPNO, specify 1 to enable the option and 0 to disable. For example, if you set XPNODATE to 1, the date is not printed at the top of each page of output.

XPNODATE	a flag that suppresses printing the date.
XPNOFOOTS	a flag that suppresses printing footnotes.
XPNOPAGE	a flag that suppresses printing the date in the header for the page.
XPNOTITLE	a flag that suppresses printing titles.

Example

This example suppresses the automatic printing of titles. You may want to do this if your report generates its own titles.

```
SAS_XPOPTST(XPNOTITLE, 1);
print your own titles with SAS_XPHDR
```

SAS_XPPRN

Print the contents of the output buffer

Usage

C	`SAS_XPPRN();`
FORTRAN	`CALL SAS_XPPRN()`
PL/I	`CALL SAS_XPPRN();`

Description

This routine is used to print the contents of the output buffer that you have built by calling SAS_XPS.

Example

This example builds a string using the SAS_XPS routine and then prints that string by calling the SAS_XPPRN routine.

```
int  offset,        /* amount to indent to center output        */
     SUM;           /* flags if extra variable should be printed */

    /* build the string */
SAS_XPS("%*tOBS %*tVARIABLE", (3+offset), (10+offset));
SAS_XPS("%*tPREVIOUS VALUE %*tNEW VALUE", (22+offset), (45+offset));
if (SUM) SAS_XPS("%*tSUM", (69+offset));

   /* put out the print buffer */
SAS_XPPRN();
```

SAS_XPRLOG

Format message associated with return code

Declarations

Type	Variable	Use	Description
rctype	*rc*	input	return code from a previous call to a SAS_XC, SAS_XD, SAS_XO, or SAS_XV routine

Usage

C	`SAS_XPRLOG(rc);`
FORTRAN	`CALL SAS_XPRLOG(rc)`
PL/I	`CALL SAS_XPRLOG(rc);`

Description

The SAS_XPRLOG routine prints error messages associated with `rctype` return codes. These types of return codes are returned by most calls to SAS_XC, SAS_XD, SAS_XO, or SAS_XV routines. The return code values all have meaningful messages associated with them that describe in detail the error that occurred. You can call SAS_XPRLOG immediately after a routine that returns an `rctype` value and SAS_XPRLOG prints the appropriate error message only if the return code indicates a warning or error. If the call to the SAS_XC, SAS_XO, or SAS_XV routine is successful, SAS_XPRLOG does nothing.

Example

```
rctype rc;
ptr    statfile;
struct XCOPSTR copn_str;

complete information in the XCOPSTR

   /* open the utility file and check return code */
if (rc = SAS_XCOPEN(&statfile, &copn_str)) {
   SAS_XPRLOG(rc);
   SAS_XEXIT(XEXITIO, 0);
}
```

SAS_XPS

Format message to the procedure output file

Declarations

Type	Variable	Use	Description
char	**formatted-string-ptr*	input	string containing control characters for substitutions. This string should be null terminated.
varies	*args*	input	substitution arguments. Refer to "Standard SAS_XP Routines" at the beginning of the discussion of the SAS_XP routines for information on substitution arguments.

Usage

C	`SAS_XPS(formatted-string-ptr, args);`
FORTRAN	`CALL SAS_XPS(formatted-string-ptr, args)`
PL/I	`CALL SAS_XPS1(formatted-string-ptr);`
	`CALL SAS_XPS2(formatted-string-ptr, arg-1);`
	`. . .`
	`CALL SAS_XPS10(formatted-string-ptr, arg-1, . . . arg-9);`

Description

This routine formats a portion of the output buffer. When you call SAS_XPS, the *formatted-string* modified by the *args*, is placed in the output buffer and the current location of the pointer to the output buffer is updated to point to the end of the string you just inserted into the buffer.

Note: SAS_XPS does not print output. To print the contents of the output buffer after it is built with SAS_XPS, call SAS_XPPRN, SAS_XPSKIP, or SAS_XPAGE. The contents of the output buffer are also printed when the procedure ends.

The SAS_XPS routine works very much like the C `printf` function. The *args* are optional and are used only when the *formatted-string* contains formatting characters, described in Table 22.6. Be sure that the formatting character you specify in the string matches the data type of the argument.

SAS_XPS *continued*

Example

```
int  SUM;          /* flags if extra variable should be printed */
char line[256] = "OBS  VARIABLE    PREVIOUS VALUE      NEW VALUE";

   /* build the string in the output buffer*/
if (SUM) SAS_XPS("%s      SUM", line);
else SAS_XPS(line);

   /* put out the print buffer and skip the next two lines */
SAS_XPSKIP(2);
```

SAS_XPSKIP

Print contents of output buffer and skip a specified number of lines

Declarations

Type	Variable	Use	Description
int	*number*	input	number of lines to skip: 0 = print contents of output buffer but do not move to the next line. That is, overprint the current line with the next line. 1 = print contents of output buffer and then move to the next line. >1 = print contents of output buffer and then move down this number of lines. Using a number greater than 1 skips *number* −1 lines.

Usage

C	`SAS_XPSKIP(number);`
FORTRAN	`CALL SAS_XPSKIP(number)`
PL/I	`CALL SAS_XPSKIP(number);`

Description

The SAS_XPSKIP routine prints the line of output stored in the output buffer and moves down the number of lines specified. If the number of lines left on the page is less than the number to skip, the routine finishes the page, but it does not skip lines at the top of the new page.

Example

Refer to the example for SAS_XPS for an illustration of this routine.

SAS_XPSLOG

Print message to the log

Declarations

Type	Variable	Use	Description
char	**formatted-string-ptr*	input	string containing control characters for substitutions. This string should be null terminated.
varies	*args*	input	substitution arguments. Refer to "Standard SAS_XP Routines" at the beginning of the discussion of the SAS_XP routines for information on substitution arguments.

Usage

C	`SAS_XPSLOG(formatted-string-ptr, args);`
FORTRAN	`CALL SAS_XPSLOG(formatted-string-ptr, args)`
PL/I	`CALL SAS_XPSLOG1(formatted-string-ptr);`
	`CALL SAS_XPSLOG2(formatted-string-ptr, arg-1);`
	`. . .`
	`CALL SAS_XPSLOG10(formatted-string-ptr, arg-1, . . . arg-9);`

Description

The SAS_XPSLOG routine is used to edit and write messages in the SAS log. Appropriate log messages are error diagnostics, completion notes, warnings, and special informatory messages. Log messages can be any length and can span several lines of your program. Text is wrapped around to subsequent lines if the message does not fit on a single line of the log. Words are never split, even at hyphens.

Note: The length of the message that you can create with SAS_XPSLOG is limited only by the length of the character string permitted on the compiler you are using. Check the documentation provided with your compiler to determine if there are limitations.

The SAS_XPSLOG routine works very much like the C `printf` function. The *args* are optional and are used only when the *formatted-string* contains formatting

SAS_XPSLOG *continued*

characters, described in Table 22.6. Be sure that the formatting character you specify in the string matches the data type of the argument.

Example

This example substitutes a short integer value into a character string and prints the message to the log.

```
int  nvar;

nvar = SFN(proc.head, 4);
SAS_XPSLOG("The number of variables in the VAR statement in %d.",
        nvar);
```

SAS_XPSPRN

Print message to the procedure output file

Declarations

Type	Variable	Use	Description
char	**formatted-string-ptr*	input	string containing control characters for substitutions. This string should be null terminated.
varies	*args*	input	substitution arguments. Refer to "Standard SAS_XP Routines" at the beginning of the discussion of the SAS_XP routines for information on substitution arguments.

Usage

C	`SAS_XPSPRN(formatted-string-ptr, args);`
FORTRAN	`CALL SAS_XPSPRN(formatted-string-ptr, args)`
PL/I	`CALL SAS_XPSPRN1(formatted-string-ptr);`
	`CALL SAS_XPSPRN2(formatted-string-ptr, arg-1);`
	`. . .`
	`CALL SAS_XPSPRN10(formatted-string-ptr, arg-1, . . . arg-9);`

Description

This routine formats a text string in the output buffer and then writes the contents of the output buffer to the procedure output file.

The SAS_XPSPRN routine works very much like the C `printf` function. The *args* are optional and are used only when the *formatted-string* contains formatting characters, described in Table 22.6. Be sure that the formatting character you specify in the string matches the data type of the argument.

Example

This example centers a string in the output buffer.

```
int  CENTER,        /* flags if output should be centered       */
     rc,
     offset,        /* amount to indent to center output        */
     SUM;           /* flags if extra variable should be printed */

CENTER = SAS_XPOPTGT(XPCENTER, &rc);

      /* build the string */
if (CENTER) {
   SAS_XPS("%*tOBS %*tVARIABLE", (3+offset), (10+offset));
   SAS_XPS("%*tPREVIOUS VALUE %*tNEW VALUE", (22+offset), (45+offset));
   if (SUM) SAS_XPS("%*tSUM", (69+offset));
}

   /* put out the print buffer */
SAS_XPPRN();
```

SAS_XPSSTR

Format message to a string

Declarations

Type	Variable	Use	Description
char	**destination-ptr*	input	destination for the formatted message.
int	*length*	input	length of the destination area.
char	**formatted-string-ptr*	input	string containing control characters for substitutions.
varies	*args*	input	substitution arguments. Refer to "Standard SAS_XP Routines" at the beginning of the discussion of the SAS_XP routines for information on substitution arguments.
int	*bytes-moved*	returned	number of bytes moved to the *destination*.

SAS_XPSSTR *continued*

Usage

C
```
bytes-moved = SAS_XPSSTR(destination-ptr, length, formatted-string-ptr,
                          args);
```

FORTRAN
```
bytes-moved = SAS_XPSSTR(destination-ptr, length, formatted-string-ptr,
                          args)
```

PL/I
```
bytes-moved = SAS_XPSSTR1(destination-ptr, length, formatted-string-ptr);
bytes-moved = SAS_XPSSTR2(destination-ptr, length, formatted-string-ptr,
                           arg-1);
. . .
bytes-moved = SAS_XPSSTR10(destination-ptr, length, formatted-string-ptr,
                            arg-1, . . . arg-9);
```

Description

This routine formats a text string to another character string. This routine is useful if you need to format a string that you do not want to print. You can call SAS_XPSPRN to print the string formatted with SAS_XPSSTR, but it is not necessary.

The SAS_XPSSTR routine works very much like the C `sprintf` function. The *args* are optional and are used only when the *formatted-string* contains formatting characters, described in Table 22.6. Be sure that the formatting character you specify in the string matches the data type of the argument.

Example

Refer to the example of SAS_XPHDR for an illustration of how to use this routine.

SAS_XS Routine: Parsing

SAS_XSPARSE is the only parsing routine in SAS/TOOLKIT software.

SAS_XSPARSE

Parse an entire procedure block of statements

Declarations

Type	Variable	Use	Description
ptr	*grammar-function-ptr*	input	grammar function return value
int	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc = SAS_XSPARSE(grammar-function-ptr, NULL, &proc);
```

Description

The SAS_XSPARSE routine parses the user's input according to the rules defined in your grammar. You must parse the statements entered by the user before you can perform most of the actions of your procedure.

Be sure that the *grammar-function* you specify as the first argument to this routine is the same as the value you specified for the GRAMFUNC= argument in the PROC USERPROC step that created the grammar function.

The second argument to this routine is always NULL and the third argument is always `&proc`. The third argument is the address of the procedure parsing structure. The parser fills information into a series of statement structures. Then it stores the address of the first statement structure in `proc.head`. All other statement structures are chained to the first one, the procedure's statement structure. You must be sure to pass the third parameter so the address of the statement structures can be stored in `proc.head`.

Example

This example illustrates how to call the SAS_XSPARSE routine with the pointer returned by your grammar.

```
ptr mygramc  U_PARMS((void));

UWPRCC(&proc);
SAS_XSPARSE(mygramc(), NULL, &proc);

if (proc.error >= XEXITPROBLEM)
   SAS_XEXIT(XEXITSYNTAX, 0);
```

SAS_XV Routines: Variable Lists and Names

Use the SAS_XV routines to obtain information about the variables the user specified in list statements, such as the VAR and BY statements. The information specified by the user is stored by the parser in a series of statement structures. Your procedure uses the semantic actions in the grammar and the SAS_XVDFLST routine to create lists of variables. You then call the SAS_XVNAME routine to obtain information about each of the variables in the list. Using this information, you call the SAS_XVGET*x* and SAS_XVPUT*x* routines to read and write observations. The process of reading observations from a SAS data set is called *scatter reading*. Writing to a SAS data set is called *gather writing*. Refer to Chapter 10 for more information on these processes.

The SAS_XVMISS and SAS_XVNAMEI routines help you create new variables to add to an output data set. Refer to the description of the STMT (statement structure) in Chapter 20 for more information on this process.

The SAS_XV routines permit you to access SAS data sets using only those variables specified in the lists. These routines automatically handle data set options like DROP= and KEEP=.

Variable Input Using SAS_XVGETx Routines

There are two kinds of SAS_XVGET routines that obtain values of variables. The scatter-input routines (XVGET*x*) can obtain many values in one call, but they involve several calls to initialize and define the process. In most cases, your procedure uses the SAS_XVGET*x* routines as follows:

- SAS_XVGETI initializes the scatter-reading process.
- SAS_XVGETD establishes where to store variables when your program reads an observation.
- SAS_XVGETE signals the end of the SAS_XVGETD calls.
- SAS_XVGET performs the scatter read. That is, it moves an observation from the input buffer to the fields defined in your program.

The SAS_XVGET1*x* routines get one value at a time. They require no initialization, but they must be called repeatedly. Therefore, if you want to get values for more than one variable, these routines are much slower than the SAS_XVGET*x* routines.

Variable Output Using SAS_XVPUTx Routines

Use the SAS_XVPUTx routines and the SAS_XOADD routine to gather write observations to a SAS data set. These routines correspond to the SAS_XVGETx input routines. In most cases, your procedure uses the SAS_XVPUTx routines as follows:

- SAS_XVPUTI initializes the gather-writing process.
- SAS_XVPUTD establishes the location in your program of variables that are to be written to the SAS data set.
- SAS_XVPUTE signals the end of the SAS_XVPUTD calls.
- SAS_XVPUT gathers the variables into an observation buffer.
- SAS_XOADD writes the observation to the output data set.

SAS_XVDFLST

Define a default list of variables

Declarations

Type	Variable	Use	Description
ptr	*file-ptr*	input	fileid of an open input data set
int	*var-type*	input	the type of data to input: 1= numeric variables 2= character variables 3= mixed (numeric/character)
struct STLIST	**field-list-ptr*	output	the statement field list where the default variable list is being built
int	*arg-count*	input	the number of arguments following this argument (0 to 20)
struct STLIST	**exclude-list-ptrs*	input	pointers to 0 or more statement field lists that contain variables to be excluded when building the default list
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

SAS_XVDFLST *continued*

Usage

```
C          rc = SAS_XVDFLST(file-ptr, var-type, field-list-ptr, arg-count,
                     <exclude-list-ptrs>);
FORTRAN    rc = SAS_XVDFLST(file-ptr, var-type, field-list-ptr, arg-count,
                     <exclude-list-ptrs>)
PL/I       rc = SAS_XVDFLST4(file-ptr, var-type, field-list-ptr, 0);
           rc = SAS_XVDFLST5(file-ptr, var-type, field-list-ptr, 1,
                     exclude-list-ptr-1);
           . . .
           rc = SAS_XVDFLST10(file-ptr, var-type,  field-list-ptr, 6,
           exclude-list-ptr-1, . . . , exclude-list-ptr-6);
```

Description

The SAS_XVDFLST routine creates a default list when the user does not explicitly specify variables for a statement. For example, if the user does not use a VAR statement in a procedure, you can create a default list of variables by using this routine. SAS_XVDFLST allows you to create default lists of all variables in the data set, or only the numeric variables, or only the character variables. You can also create default lists that exclude variables in other lists. For example, if the user specifies a variable in a BY statement, you probably do not want the variable to be repeated in the default list of variables for the VAR statement. If you do not need to exclude any other lists, specify 0 for *arg-count* and omit the *exclude-lists*.

Most procedures that permit a VAR statement should create a default VAR list when the user does not specify the VAR statement. In most cases, you need to create a default list for only the VAR statement.

If you call SAS_XVDFLST for a list that already exists (because the user specified that statement), SAS_XVDFLST has no effect. Therefore, you do not need to test if the list already exists before calling SAS_XVDFLST.

The default list does not have to be the same type of list as the one that can be specified by the user. The default list can be more limited. For example, you can permit users to enter character or numeric variables, but create a default list of only the numeric variables.

Note: For PL/I programs you must use a numbered version of SAS_XVDFLST. The routine you use depends on how many *exclude-list-ptr* arguments you specify. If you specify one pointer, use SAS_XVDFLST5 because the total number of arguments for the routine is 5.

Example

In this example, there are four possible lists. Each list is assigned a pointer. The example then calls SAS_XVDFLST to create a default list for the first list, which is numeric only. If list 1 is already defined because the user specified the statement associated with this list, SAS_XVDFLST has no effect. If list 1 is not defined, then SAS_XVDFLST builds list 1 out of all the numeric variables that are not in lists 4, 5, or 6.

```
struct STLIST *list1,
              *list4,
              *list5,
              *list6;
ptr    inputfid;

   /* parse the lists */
list1 = (struct STLIST *)SFLD(proc.head, 1);
list4 = (struct STLIST *)SFLD(proc.head, 4);
list5 = (struct STLIST *)SFLD(proc.head, 5);
list6 = (struct STLIST *)SFLD(proc.head, 6);

   /* create a default list for list1 */
SAS_XVDFLST(inputfid, 1, list1, 3, list4, list5, list6);
```

SAS_XVFIND

Find a variable within a data set

Declarations

Type	Variable	Use	Description
ptr	*file-ptr*	input	fileid of an open input data set
char8	*name*	input	name of variable to be located
short	**var-number-ptr*	output	number of variable within the data set
rctype	*rc*	returned	return code: 0 = successful nonzero = variable not found

Usage

```
rc = SAS_XVFIND(file-ptr, name, var-number-ptr);
```

Description

The SAS_XVFIND routine allows you to determine if a variable exists in a data set. SAS_XVFIND is particularly useful if your procedure requires that a specific variable is present.

Note: The *var-number-ptr* must be a pointer to a short integer, not an `int` or a `long`.

SAS_XVFIND *continued*

Example

If you want to determine if a specific variable has been specified in a variable list, first call SAS_XVFIND with the name of the variable. Compare the *var-number* returned by SAS_XVFIND with each of the variable numbers in the `ilist` array of the FIELDSTR for the list. The following example illustrates this technique:

```
ptr    infile;
short  found,
       nvar,
       var_num,
       *list,
       x;
rctype rc;

   /* access the list in the statement structure */
list = (short *)SFLP(proc.head, 4);
nvar = SFN(proc.head, 4);

   /* get variable number for specific variable */
rc = SAS_XVFIND(infile, "SSN     ", var_num);
XPRLOG(rc);

   /* test variables in list to find a match */
for (x = 0, found = 0; x < nvar && found == 0; x++) {
   if (var_num == list[x]) found = 1;
}
```

SAS_XVGET

Execute scatter read

Declarations

Type	Variable	Use	Description
ptr	*xvget-ptr*	input	control pointer returned by SAS_XVGETI.
ptr	*base-ptr*	input	base address to be used with offset definition. This pointer should be NULL if the call to SAS_XVGETD specified locations instead of offset values.
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful.

Usage

```
rc = SAS_XVGET(xvget-ptr, base-ptr);
```

Description

The SAS_XVGET routine moves an observation from the input buffer to the program's storage areas; that is, it performs the scatter-input process.

As the values of the variables are moved from the input buffer to the locations defined by SAS_XVGETD, the lengths of the source and destination field may not match. In this case, SAS_XVGET pads or truncates the values as necessary. Note the following rules used when the lengths do not match:

- If the source length is less than the destination length, the destination value is padded. Unformatted characters are padded to the right with blanks, and unformatted numbers are padded using host-specific routines.
- If the source length is greater than the destination length, the source value is truncated when it is stored in the destination. Unformatted characters are truncated from the right, and unformatted numbers are truncated using host-specific routines.

Before you can use this routine, you must do all of the following:

- Initialize the input process by using SAS_XVGETI.
- Establish where to store the fields of each observation by calling SAS_XVGETD.
- Complete the scatter-input definition by calling SAS_XVGETE.
- Read an observation from the data set into the input buffer by calling SAS_XBYGET or SAS_XOGETN.

You can use this routine in two ways, depending upon how you call SAS_XVGETD. In the calls to SAS_XVGETD for each variable to be processed, you must specify either the offset from a base location where the variable should be stored or the actual pointer to the location. If you specify the offset in the call to SAS_XVGETD, you must specify the base-pointer value for the *base-ptr* argument in the call to SAS_XVGET. If you specify the actual location in the calls to SAS_XVGETD, specify NULL for *base-ptr* in the call to SAS_XVGET.

The capability to specify offset values from a base pointer makes it easier to store the values from multiple observations. Instead of reading each observation into the same storage location, which occurs when you specify actual addresses in the call to SAS_XVGETD, you can create an array of observations.

The second example in the following section shows how to create an array of all observations in a BY group.

Examples

This first example illustrates how to use SAS_XVGET when you have set up the call to SAS_XVGETD with specific locations of variables (not offsets from a base address). In this example, the variables to be processed are all numeric and pointed to by `list`. The number of variables is stored in `nvar`. The variables are moved to the array, `observat`. Note that the call to SAS_XVGETD specifies the

SAS_XVGET *continued*

location in the array for each variable, so the subsequent calls to SAS_XVGET specify NULL instead of a base address.

```
ptr       infile,
          ptr_xvg;
short     nvar,
          *list;
int       x;
double    *observat;

allocate space for observations based on number of variables

   /* obtain list pointer */
list = (short *)SFLP(proc.head, 1);

   /* define scatter read with specific locations */
SAS_XVGETI(infile, nvar, &ptr_xvg);
for (x = 0; x < nvar; x++) {
   SAS_XVGETD(ptr_xvg, list[x], 0, observat+x, sizeof(double), XV_NOFMT);
}
SAS_XVGETE(infile);

   /* read observations */
while (SAS_XBYNEXT(infile) > 0) {
   while (SAS_XBYGET(infile) != NULL) {
      SAS_XVGET(ptr_xvg, NULL);
      process observation
   }
}
```

This example illustrates how to use offsets in the call to SAS_XVGETD and then specify the base address in the calls to SAS_XVGET. This example also illustrates how to determine the number of observations in a BY group and then create an array that holds all the observations in a BY group. By incrementing the address for the base pointer, the second argument for SAS_XVGET, you can create an array of observations.

Note: In this example, the variable list for the VAR statement is assigned to list 6, all variables are floating-point values, and the values for all observations in a BY group are stored in the array `obsarray`.

```
double *obsarray;
long   maxobs,
       numobs,
       obsindex,
       offset;
ptr    infileid,
       xvgetptr;
```

```
short  j,
       *list,
       nvar;

maxobs = 1;
   /* find largest BY group */
while (SAS_XBYNEXT(infileid) > 0) {
   numobs = 0;
   while (SAS_XBYGET(infileid) != NULL) {
      numobs += 1;
   }
   maxobs = (maxobs > numobs) ? maxobs : numobs;
}

   /* rewind data set for later processing */
SAS_XBYRWND(infileid, 'F');

   /* determine number of variables */
nvar = SFN(proc.head, 6);

   /* allocate space for array of observations */
obsarray = (double *)SAS_XMEMEX(maxobs * sizeof(double) * nvar);

   /* initialize input buffer */
SAS_XVGETI(infileid, nvar, &xvgetptr);

   /* set up scatter input */
for (offset = 0, j = 0; j < nvar; j++, offset += sizeof(double)) {
   SAS_XVGETD(xvgetptr, list[j], offset, NULL, sizeof(double),
              XV_NOFMT);
}
SAS_XVGETE(xvgetptr);

   /* process observations */
while (SAS_XBYNEXT(infileid) > 0) {

      /* start each BY group at the beginning */
      /* of the observation array             */
   obsindex = 0;
   while (SAS_XBYGET(infileid) != NULL) {

         /* move variables for a single observation in the BY group */
      SAS_XVGET(xvgetptr, obsarray + obsindex);
      further processing

         /* move to next observation in the array */
      obsindex += nvar;
   }
}
```

SAS_XVGETD

Define a variable for a scatter read

Declarations

Type	Variable	Use	Description
ptr	*xvget-ptr*	input	control pointer returned by SAS_XVGETI.
int	*var-number*	input	variable number within the input file.
long	*offset*	input	offset from base or 0 if this is not an offset move.
ptr	*location*	input	address of destination area or NULL for offset moves.
int	*length*	input	length of the destination area.
int	*type-move*	input	type of move being done: XV_NOFMT = unformatted read. This is the value most commonly used. XV_FMT = formatted read. Use this value when you want to read values with specified formats. If you specify formatted values, be sure to call SAS_XFFILE to obtain the formats before you call.
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful.

Usage

```
rc = SAS_XVGETD(xvget-ptr, var-number, offset, location, length, type-move);
```

Description

The SAS_XVGETD routine establishes where the value for a variable is stored when SAS_XVGET is called. You must call this routine at least once for each variable you want to access in a SAS data set. After you set up storage locations for variables, you must read the observations by calling SAS_XBYGET or SAS_XOGETN and store the variables in program storage by calling SAS_XVGET. The process of reading observations into locations established by SAS_XVGETD is called scatter input.

If SAS_XVGETD detects that it has been called incorrectly, it automatically issues an error message by calling SAS_XEBUG.

You can use this routine in two ways. In each call to SAS_XVGETD, you must specify either the offset from a base location where the variable should be stored or the actual pointer to the location. That is, you can supply a value for only one of the *offset* or *location* arguments. The other argument must be NULL.

If you specify the offset in the call to SAS_XVGETD, you must specify the base-pointer value for the *base-ptr* argument when you call SAS_XVGET. If you specify the actual location in the calls to SAS_XVGETD, specify NULL for *base-ptr* in the call to SAS_XVGET. You can mix these methods for the variables that are read in a scatter-input operation.

The capability to specify offset values from a base pointer makes it easier to store the values from multiple observations. Refer to the discussion and example for SAS_XVGET for more information on this technique.

The number of calls to this function should not be greater than the *number-of-vars* argument in the call to SAS_XVGETI.

Examples

The most common use of SAS_XVGETD is illustrated in the following example. This example also shows how SAS_XVGETD relates to SAS_XVGETI and SAS_XVGET.

Note: In this example, the variable list for the VAR statement is assigned to list 6, all variables are floating-point values, and the values are stored in the array `vararray`.

```
double *vararray;
ptr    infileid,
       xvgetptr;
short  j,
       *list,
       nvar;
long   maxobs;

maxobs = 1;

   /* initialize input buffer and get list*/
nvar = SFN(proc.head, 6);
SAS_XVGETI(infileid, nvar, &xvgetptr);
list = (short *)SFLP(proc.head, 6);

for (j = 0; j < nvar; j++) {
   SAS_XVGETD(xvgetptr, list[j], 0, vararray + j, sizeof(double),
              XV_NOFMT);
}

   /* start each BY group at the beginning */
   /* of the observation array            */
while (SAS_XBYGET(infileid) != NULL) {
   while (SAS_XBYGET(infileid) != NULL) {

         /* move variables for a single observation in the BY group */
      SAS_XVGET(xvgetptr, NULL);
      further processing
   }
}
```

SAS_XVGETD *continued*

In addition to storing input observations into an array, as in the previous example, you can divide the data for the observation and store them in several arrays or other scattered storage areas. This ability to read data into noncontiguous storage areas is the reason these routines are called *scatter-input* routines. For example, if you want to store all numeric variables in one array and all character variables in a separate array, follow the next example.

Each call to SAS_XVGET in this example automatically stores the character variables in `chararray` and the numeric variables in `fltarray`. The example determines the size of these arrays by testing the length of each variable stored in the NAMESTR for the variable. The example then allocates the necessary storage for the numeric array (the number of numeric variables times the size of a `double`) and the character array (the sum of the lengths of each character variable).

```
double *fltarray;
long   charlens;
ptr    infileid,
       xvgetptr,
       chararray;
short  c,
       j,
       *list,
       numvars,
       nvar;
struct NAMESTR *storenam;
int    n;

   /* initialize input buffer and get list*/
nvar = SFN(proc.head, 6);
SAS_XVGETI(infileid, nvar, &xvgetptr);
list = (short *)SFLP(proc.head,6);

   /* determine the total number of numeric variables */
   /* and the total lengths of character variables    */
numvars = 0;
for (j = 0; j < nvar; j++) {
   SAS_XVNAME(infileid, list[j], &storenam);
   if (storenam->ntype == 1) numvars += 1;
   else charlens += (storenam->nlng);
}

   /* allocate space for both arrays */
fltarray = (double *)SAS_XMEMEX(numvars * sizeof(double));
chararray = SAS_XMEMEX(charlens);

   /* scatter read variables to two arrays */
for (j = 0, c = 0, n = 0; j < nvar; j++) {
   SAS_XVNAME(infileid, list[j], &storenam);
```

```
      /* store character variables in the two-dimensional chararray */
   if (storenam->ntype == 2) {
      SAS_XVGETD(xvgetptr, list[j], 0, chararray + c, storenam->nlng,
                 XV_NOFMT);
      c += storenam->nlng;
   }

      /* store numeric variables in the floating-point array */
   else {
      SAS_XVGETD(xvgetptr, list[j], 0, fltarray + n, sizeof(double),
                 XV_NOFMT);
      n += 1;
   }
}
SAS_XVGETE(xvgetptr);

   /* process observations */
while (SAS_XBYGET(infileid) != NULL) {
   while (SAS_XBYGET(infileid) != NULL) {
      SAS_XVGET(xvgetptr, NULL);
      further processing
   }
}
```

Note: The process of scatter reading is illustrated in Figure 10.1 and discussed in more detail in Chapter 10.

SAS_XVGETE

Terminate scatter-input definition

Declarations

Type	Variable	Use	Description
ptr	*xvget-ptr*	input	control pointer returned by SAS_XVGETI
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc = SAS_XVGETE(xvget-ptr);
```

SAS_XVGETE *continued*

Description

Call SAS_XVGETE to indicate that you have finished defining the scatter-input process. You must call this routine after all of the SAS_XVGETD calls and before the first SAS_XVGET call.

Example

Refer to the examples for SAS_XVGETD for an illustration of how to use SAS_XVGETE.

SAS_XVGETI

Initialize scatter-read definitions

Declarations

Type	Variable	Use	Description
ptr	*file-ptr*	input	fileid of an open input data set.
int	*number-of-vars*	input	number of variables being moved. This parameter should be at least as large as the number of SAS_XVGETD calls that follow.
ptr	**ptr-to-xvget-ptr*	output	pointer to the control pointer for all other SAS_XVGETx calls.
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful (usually caused by insufficient memory).

Usage

```
rc = SAS_XVGETI(file-ptr, number-of-vars, ptr-to-xvget-ptr);
```

Description

The SAS_XVGETI routine initializes the process required to read observations from a SAS data set. Before you can use any other SAS_XVGETx routine, you must initialize the input process. This routine sets the *xvget-ptr* parameter, which must be used in subsequent SAS_XVGETx calls.

▶ ***Caution .... Do not use the xvget-ptr in calls to SAS_XVPUT routines.*** ▲

After you call SAS_XVGETI, you then call SAS_XVGETD for each variable that will be read into the input buffer. After all calls to SAS_XVGETD are completed, call SAS_XVGETE to end the definition phase. To actually read the data set, first call SAS_XBYGET to read an observation into the input buffer and follow it with a call to SAS_XVGET to move the variables from the buffer area to the storage areas where they can be accessed by your program. You do not directly access observations in the buffer.

The variables in the scatter input can come from several lists. The number specified in the *number-of-variable* parameter must be greater than or equal to the number of calls that are made to SAS_XVGETD so that sufficient memory is allocated for the input buffer. Use SAS_XOINFO to obtain the number of variables in a list.

Note: Do not call SAS_XVGETI if you are reading observations by using only the SAS_XVGET1*x* routines.

Example

Refer to the examples for SAS_XVGETD for an illustration of how to use SAS_XVGETI.

SAS_XVGETT

Terminate scatter-input processing

Declarations

Type	Variable	Use	Description
ptr	*xvget-ptr*	input	control pointer returned by SAS_XVGETI
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc = SAS_XVGETT(xvget-ptr);
```

Description

The SAS_XVGETT routine frees the memory allocated by the SAS_XVGETI routine. If you need to free this memory, you must use SAS_XVGETT; do not use SAS_XMEMFRE.

Note: All memory allocated for your procedure is freed when the procedure finishes executing; therefore, in most cases it is not necessary to free memory allocated by SAS_XVGETI. If you want to free this memory while your procedure

SAS_XVGETT *continued*

is still executing, you can call SAS_XVGETT after all SAS_XVGET calls are completed.

Do not attempt to use *xvget-ptr* after you call SAS_XVGETT.

Example

This example illustrates calling SAS_XVGETT to free the memory used by the SAS_XVGET routines. Note that you can call SAS_XVGETT only after the loop that reads the observations from the data set.

```
short  nvar,
       *list;
int    j;
ptr    infileid,
       xvgetptr;
double vararray[];

   /* initialize input buffer and get list */
nvar = SFN(proc.head, 6);
SAS_XVGETI(infileid, nvar, &xvgetptr);
list = (short *) SFLP(proc.head, 6);

for (j = 0; j < nvar; j++) {
   SAS_XVGETD(xvgetptr, list[j], 0, vararray + j, sizeof(double),
              XV_NOFMT); }
while (SAS_XBYGET(infileid) != NULL) {
   while (SAS_XBYGET(infileid) != NULL) {

         /* move variables for a single observation in the BY group */
      SAS_XVGET(xvgetptr, NULL);
      further processing
   }
}
SAS_XVGETT(xvgetptr);
further processing that does not require reading from
the input data set
```

SAS_XVGET1C

Get a character variable's value from the input observation buffer

Declarations

Type	Variable	Use	Description
ptr	*file-ptr*	input	fileid of an open input data set
int	*var-number*	input	variable number
int	*length*	input	length of the destination area
ptr	*string*	output	destination for character value
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc = SAS_XVGET1C(file-ptr, var-number, length, string);
```

Description

The SAS_XVGET1C routine gets the value of a single character variable from a SAS data set observation. Before calling SAS_XVGET1C, you must call SAS_XOGETN or SAS_XBYGET to read the observation from the data set into the input observation buffer. You do not need to use the SAS_XVGETI routine with SAS_XVGET1C.

If the length of the destination area is less than the actual length of the character variable, the value is truncated from the right. If the length of the destination area is greater than the length of the character data being moved, then the destination area is padded on the right with blanks.

Note: Repeated calls to this routine use much more CPU time than calls to SAS_XVGET. It is more efficient to use the SAS_XVGETx routines if you are reading the value of more than one variable. This entry provides a short cut method for reading a single character variable.

Example

This example reads only one character variable from a file.

```
ptr   onevards;
short *list;
int   rc,
      varnum;
char  string[200];
```

SAS_XVGET1C *continued*

```
   /* check to see if file was specified */
onevards = SFFILE(proc.head, 1);
list = (short *)SFLP(proc.head, 4);
varnum = (int) *list;
if (onevards != NULL && varnum != 0) {

      /* loop through observations */
   while (SAS_XBYGET(onevards) != NULL)    {

         /* read single character variable */
      rc = SAS_XVGET1C(onevards, varnum , 200, string );
      SAS_XPSLOG("The variable value is %*s", SAS_ZSTRIP(string,200),
                  string);
   }
}
```

SAS_XVGET1F

Get a numeric variable's value from the input observation buffer

Declarations

Type	Variable	Use	Description
ptr	*file-ptr*	input	fileid of an open input data set
int	*var-number*	input	variable number
double	**value-ptr*	output	variable value
rctype	*rc*	returned	return code:
			0 = successful
			nonzero = unsuccessful

Usage

```
rc = SAS_XVGET1F(file-ptr, var-number, value-ptr);
```

Description

The SAS_XVGET1F routine gets the value of a single numeric variable from a SAS data set observation. Before calling SAS_XVGET1F, you must call SAS_XOGETN or SAS_XBYGET to read the observation from the data set into the input observation buffer. You do not need to use the SAS_XVGETI routine with SAS_XVGET1F.

If the variable being moved is a shortnumeric (2-7 bytes), the result is padded into the destination using host-specific techniques.

Note: Repeated calls to this routine use much more CPU time than calls to SAS_XVGET. It is more efficient to use the SAS_XVGETx routines if you are reading the value of more than one variable. This entry provides a short cut method for reading a single numeric variable.

Example

This example reads a numeric variable from an input data set. The example reads the first variable listed in the VAR statement (list 4, in this case).

```
double   weight;
int      rc,
         varnum;
ptr      onevards;
short    *list;

   /* check to see if file was specified */
onevards = SFFILE(proc.head, 2);
list = (short *)SFLP(proc.head, 4);
varnum = (int) *list;
if (onevards != NULL) {

      /* loop through observations */
   while (SAS_XBYGET(onevards) != NULL)   {

         /* read single variable */
      rc = SAS_XVGET1F(onevards, varnum , &weight);
      weight = SAS_ZMISS(weight) ? 0 : (int)weight;
   }
}
```

SAS_XVGET1V

Get a formatted value from a variable in the input observation buffer

Declarations

Type	Variable	Use	Description
ptr	*file-ptr*	input	fileid of an open input data set
int	*var-number*	input	variable number
int	*length*	input	length of the destination area
ptr	*string-ptr*	output	destination for character value
rctype	*rc*	returned	return code:
			0 = successful
			nonzero = unsuccessful

SAS_XVGET1V *continued*

Usage

```
rc = SAS_XVGET1V(file-ptr, var-number, length, string-ptr);
```

Description

The SAS_XVGET1V routine gets the value of a single formatted variable from a SAS data set observation. Before calling SAS_XVGET1V, you must call SAS_XOGETN or SAS_XBYGET to read the observation from the data set into the input observation buffer. You do not need to use the SAS_XVGETI routine with SAS_XVGET1V.

You must call SAS_XFFILE before you call this function. Otherwise, the values are formatted using the default format rather than user-specified formats. If no format is specified, numeric values are formatted using the BEST. format with the length specified in the *length* argument. If the length of the specified format is different from the length of the string, the value is truncated or padded with blanks.

Note: Repeated calls to this routine use much more CPU time than calls to SAS_XVGET. It is more efficient to use the SAS_XVGETx routines if you are reading the value of more than one variable. This entry provides a short cut method for reading a single formatted variable.

Example

This example reads a formatted numeric variable from an input data set. The example reads the first variable listed in the VAR statement (list 4, in this case).

```
int   rc,
      varnum;
short *list;
ptr   onevards;
char  formatted[32];

   /* check to see if file was specified */
onevards = SFFILE(proc.head, 2);
list = (short *)SFLP(proc.head, 4);
varnum = (int) *list;
if (onevards != NULL) {

      /* load formats for file */
   SAS_XFFILE(onevards, 'F', list, 1);

      /* loop through observations */
   while (SAS_XBYGET(onevards) != NULL)   {

         /* read single variable */
      rc = SAS_XVGET1V(onevards, varnum, 32, &formatted);
   }
}
```

SAS_XVMISS

Set all variable values to missing in a buffer

Declarations

Type	Variable	Use	Description
ptr	*file-ptr*	input	fileid of an open input data set
ptr	*miss-ptr*	input	pointer to the program observation buffer allocated in your program or NULL to initialize the output data set buffer
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc = SAS_XVMISS(file-ptr, miss-ptr);
```

Description

Use the SAS_XVMISS routine to get a missing value buffer if you need to fill an observation with missing values. The size and type of missing values for the observation depend upon attributes of the data set. That is, all character variables in the observation buffer are set to blanks. All numeric variables are set to missing. Do not attempt to create a buffer of missing values by setting the values yourself; use this routine instead.

Before calling this routine with a *miss-ptr* other than NULL, you must allocate a block of memory that is at least as long as the observation buffer and then store the address of this memory block in *miss-ptr*. Obtain the length of an observation by calling SAS_XOINFO with the XO_LREC code. You can call SAS_XVMISS before calling the SAS_XVGETI routine.

If you specify NULL for the *miss-ptr* argument, SAS_XVMISS initializes the output buffer used by SAS_XVPUT to missing values.

Example

This example illustrates calling SAS_XVMISS with NULL for the second argument, which changes the values in the buffer used by SAS_XVPUT to missing values. Once you have initialized these values to missing, you can then set the values of some of the variables if needed.

SAS_XVMISS *continued*

```
ptr       infileid,
          outfileid,
          xvgetptr,
          xvputptr;

set up scatter reads and gather writes

   /* process observations */
while (SAS_XBYGET(infileid) != NULL) {

      /* move variables for a single observation in the BY group */
   SAS_XVGET(xvgetptr, NULL);
   test value of variables.  Determine if an observation needs
   to be rebuilt.

      /* initialize the output observation to missing values */
   SAS_XVMISS(infileid, NULL);
   set values of variables as needed

      /* write observation to output buffer */
   SAS_XVPUT(xvputptr, NULL);

      /* write output buffer to data set */
   SAS_XOADD(outfileid, xvputptr);
}
```

SAS_XVNAME

Fetch the NAMESTR for a selected variable

Declarations

Type	Variable	Use	Description
ptr	*file-ptr*	input	fileid of an open input data set
int	*var-number*	input	variable number
nameptr	**ptr-to-namestr-ptr*	output	pointer to the address of the NAMESTR for a variable in the input data set
rctype	*rc*	returned	return code: 0 = successful nonzero = variable not found

Usage

```
rc = SAS_XVNAME(file-ptr, var-number, ptr-to-namestr-ptr);
```

Description

The SAS_XVNAME routine provides name and attribute information for variables. When you call SAS_XVNAME, the routine accesses the variable name structures (NAMESTR) that are created when a data set is opened for input. A data set can be opened for input by the @DS or the @DSDFLT semantic action in your grammar or by a call to SAS_XOOPEN. Either method creates a NAMESTR for each variable in a data set opened for input. Variable name structures for output data sets must be created by your procedure.

In most cases, you need to call SAS_XVNAME to get information about the variable so you can determine how to process it. Note that you should also follow the call to SAS_XVNAME with a call to SAS_XFFILE, which adds information about formats and informats to the NAMESTR for each variable.

▶ *Caution* .... ***You must not change any of the values in the NAMESTR for an input data set.*** You must create a new variable name structure, copy the contents of the variable's NAMESTR from the input data set, and change the values in the copy. Use the SAS_XVNAMEI routine to initialize the new NAMESTR. Then in the gather output process, when you call SAS_XVPUTD, be sure to point to the NAMESTR you have created instead of the NAMESTR from the input data set. ▲

The last argument to this function, *ptr-to-namestr-ptr*, must provide the address of a pointer to a NAMESTR data structure. Do not simply provide a pointer to a NAMESTR; you must supply the address of the pointer. This enables SAS_XVNAME to provide the location of the NAMESTRs for the input data set.

Example

This example illustrates calling SAS_XVNAME to obtain information about variables before writing the variables to the output data set. The example calls SAS_XFFILE to add format information to the NAMESTRs. The numeric variables from the input data set are stored in the `observat` array of doubles. The `list` array contains the variable numbers from the VAR list. Refer to the example for SAS_XVPUTI for an illustration of creating a new variable.

```
double         *observat;
ptr            infile,
               out_ptr;
short          *list,
               nvar;
int            x;
struct NAMESTR *nam_ptr;

access fileid and variable list

   /* add format information to NAMESTRs */
SAS_XFFILE(infile, 'F', list, nvar);
```

SAS_XVNAME *continued*

```
set up scatter reads

   /* set up gather writes */
for (x = 0; x < nvar; x++) {
   SAS_XVNAME(infile, list[x], &nam_ptr);
   SAS_XVPUTD(out_ptr, nam_ptr, 0, observat+x, sizeof(double));
}

SAS_XVPUTE(out_ptr);
```

In addition, refer to the second example for SAS_XVGETD for an illustration of how to use the type and length information in the NAMESTR to create two arrays to store character and numeric variables.

SAS_XVNAMEI

Initialize the fields of a NAMESTR structure

Declarations

Type	Variable	Use	Description
nameptr	*ptr-to-namestr*	input	pointer to the NAMESTR to initialize
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc = SAS_XVNAMEI(ptr-to-namestr);
```

Description

The SAS_XVNAMEI routine initializes the fields of the variable name structure (NAMESTR). The variable name, variable label, format name, and informat name fields are set to blanks. All other fields are set to 0.

Before you call SAS_XVNAMEI, you must create a new variable name structure. If you want to reuse a NAMESTR for several invocations of SAS_XVPUTD, you can also use SAS_XVNAMEI to reinitialize the storage area.

After you call SAS_XVNAMEI you must complete the following fields in the NAMESTR:

- **nname** = variable name
- **ntype** = variable type
- **nlng** = variable length.

Note: This routine does not create a NAMESTR; it clears an existing structure.

Example

Refer to the example for SAS_XVNAME for an illustration of how to use SAS_XVNAMEI to initialize a NAMESTR for a new variable.

SAS_XVPUT

Build or update an observation

Declarations

Type	Variable	Use	Description
ptr	*xvput-ptr*	input	control pointer returned by SAS_XVPUTI.
ptr	*offset*	input	base address to be used with offset definition. This pointer may be NULL if offset specfications are not used.
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful.

Usage

```
rc = SAS_XVPUT(xvput-ptr, offset);
```

Description

The SAS_XVPUT routine gathers the variable values from the locations defined by SAS_XVPUTD and creates the output observation in a buffer. You must then call SAS_XOADD to write the observation to the output file. Before you can use this routine, you must initialize the output process by using SAS_XVPUTI and establish where to find the variable values for each observation by calling SAS_XVPUTD.

As the values of the variables are moved from the program's storage areas to the output buffer, the lengths of the source and destination field may not match.

SAS_XVPUT *continued*

In this case, SAS_XVPUT pads or truncates the values as necessary. The following rules are used when the lengths do not match:

- If the source length is less than the destination length, the destination value is padded. Character values are padded to the right with blanks, and numbers are padded using host-specific routines.
- If the source length is greater than the destination length, the source value is truncated when it is stored in the destination. Character values are truncated from the right, and numbers are truncated using host-specific entries.

Example

This example illustrates how to initialize the scatter-read and gather-write process when the output file has the same variables as the list of variables processed in the input file. Only numeric variables are being processed in this example. The `list` array contains the variable numbers of the variables selected from the input data set. The `observat` array is being defined to store the values of the numeric variables.

```
double *observat;
ptr    infile,
       outfile,
       ptr_xvget,
       ptr_xvput;
short  *list,
       nvar,
       x;
struct NAMESTR *namptr;

   /* access the variable list and number of variables */
list = (short *)SFLP(proc.head, 6);
nvar = SFN(proc.head, 6);

   /* allocate memory for storing observation */
observat = (double *)SAS_XMEMEX((long)nvar * sizeof(double));

   /* define scatter reads and gather writes with specific locations */
SAS_XVGETI(infile, nvar, &ptr_xvget);
SAS_XVPUTI(outfile, nvar, &ptr_xvput);
for (x = 0; x < nvar; x++) {
   SAS_XVGETD(ptr_xvget, list[x], 0, observat+x,
              sizeof(double), XV_NOFMT);
   SAS_XVNAME(infile, list[x], &namptr);
   SAS_XVPUTD(ptr_xvput, namptr, 0, observat+x, sizeof(double));
}
SAS_XVGETE(ptr_xvget);
SAS_XVPUTE(ptr_xvput);
```

```
   /* read observations */
while (SAS_XBYNEXT(infile) > 0) {
   while (SAS_XBYGET(infile) != NULL) {
      SAS_XVGET(ptr_xvget, NULL);
      process observation
      SAS_XVPUT(ptr_xvput, NULL);
      SAS_XOADD(outfile, ptr_xvput);
   }
}
```

SAS_XVPUTCV

Copy the specifications of variables from an input data set to an output data set

Declarations

Type	Variable	Use	Description
ptr	*xvput-ptr*	input	control pointer returned by SAS_XVPUTI
ptr	*file-ptr*	input	fileid of an open input data set
short	**ptr-to-var-list*	input	pointer to the list of variable numbers or NULL for all variables
int	*number-of-vars*	input	number of variables in list
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc = SAS_XVPUTCV(xvput-ptr, file-ptr, ptr-to-var-list, number-of-vars);
```

Description

The SAS_XVPUTCV routine provides a short-cut for adding variables to an output data set from the input data set. For example, in many cases, procedures permit both a BY statement and a VAR statement. The procedure may process the variables in the VAR statement before putting them in the output data set. In most cases, no processing is performed on the variables in the BY list, but the procedure still needs to add those variables to the output data set. For the variables in the VAR statement, you use the SAS_XVGETI, SAS_XVGETD, SAS_XVGETE, and SAS_XVGET routines to scatter read the values. Then you use the SAS_XVPUTI, SAS_XVPUTD, SAS_XVPUTE, and SAS_XVPUT routines to gather write the variables after processing. To handle the BY variables, use SAS_XVPUTCV, which simply adds the variables from the input data set to the output data set without setting up a scatter-read, gather-write process.

SAS_XVPUTCV *continued*

Note that you are not limited to using SAS_XVPUTCV for only BY-variable lists. In fact, you can use SAS_XVPUTCV for any list of variables that can be moved directly from the input data set to the output data set.

When you use SAS_XVPUTCV for a list of variables, you do not need to call SAS_XVPUTD for those variables. You must, however, count those variables in the number of variables passed in the call to SAS_XVPUTI.

Example

This example illustrates how to add variables that are not changed in the program to the output data set. The variables that are being changed by the program are in list 6 in this example. The call to SAS_XVPUTI includes in the number of variables all the variables in list 6 as well as the variables in the BY statements (list 0). Note that the call to SAS_XVDFLST excludes the variables in the BY statement when building the default list. The variables in the BY statement do not need to be processed in the same way as the variables in list 6. After the scatter-read and gather-write processes are completed for list 6, a later call to SAS_XVPUTCV adds the variables from list 0 to the output data set.

```
double *observat;
ptr    infile,
       outfile,
       ptr_xvget,
       ptr_xvput;
short  nbyvar,
       nvar,
       *varlist,
       x;
struct NAMESTR *namptr;
struct STLIST *bylist,
              *list;

   /* access the variable list and BY list */
list = (struct STLIST *)SFLD(proc.head, 6);
bylist = (struct STLIST *)SFLD(proc.head, 0);
varlist = (short *)SFLP(proc.head, 6);

   /* set up default list of all numeric variables if list 6 is empty */
SAS_XVDFLST(infile, 1, list, 1, bylist);

   /* find out how many variables are in list 6 and BY list */
nvar = SFN(proc.head, 6);
nbyvar = SFN(proc.head, 0);

allocate memory for storing the observation

   /* initialize scatter reads for variables that change */
SAS_XVGETI(infile, nvar, &ptr_xvget);
```

```
   /* initialize gather writes for all variables */
SAS_XVPUTI(outfile, nvar+nbyvar, &ptr_xvput);

   /* define scatter reads and gather writes */
   /* for variables that change              */
for (x = 0; x < nvar; x++) {
   SAS_XVGETD(ptr_xvget, varlist[x], 0, observat+x,
              sizeof(double), XV_NOFMT);
   SAS_XVNAME(infile, varlist[x], &namptr);
   SAS_XVPUTD(ptr_xvput, namptr, 0, observat+x, sizeof(double));
}

   /* scatter read and gather write not needed */
   /* for variables that do not change         */
SAS_XVPUTCV(ptr_xvput, outfile, bylist, nbyvar);

   /* end of calls to SAS_XVGETD and SAS_XVPUTD */
SAS_XVGETE(ptr_xvget);
SAS_XVPUTE(ptr_xvput);
```

SAS_XVPUTD

Define the gather-write operation for a variable

Declarations

Type	Variable	Use	Description
ptr	*xvput-ptr*	input	control pointer returned by SAS_XVPUTI.
nameptr	*ptr-to-namestr*	input	pointer to the NAMESTR structure for the variable to be moved.
long	*offset*	input	offset of source data or 0 for non-offset moves.
ptr	*location*	input	address of data to be written to the observation buffer. This argument should be NULL for offset moves.
int	*length*	input	length of source data or 0 for default.
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful.

Usage

```
rc = SAS_XVPUTD(xvput-ptr, ptr-to-namestr, offset, location, length);
```

SAS_XVPUTD *continued*

Description

The SAS_XVPUTD routine establishes where to find the values for variables when an output observation buffer is built. Call this routine once for each variable your program changes before writing it to a SAS data set. (Use SAS_XVPUTCV for variables that do not change.)

The process of creating observations from data stored in noncontiguous memory is called *gather output*. The order in which you define the output variables determines the order in which they appear within the data set.

After you define the gather-output operation for all variables, you must move the observations to the output buffer by calling SAS_XVPUT and write the output record with SAS_XOADD.

You can use this routine in two ways. In each call to SAS_XVPUTD, you must specify either the offset from a base location where the variable is stored or the actual pointer to the location. That is, you can supply a value for only one of the *offset* or *location* arguments. The other argument must be NULL.

If you specify the offset in the call to SAS_XVPUTD, you must specify the base-pointer value for the *base-ptr* argument when you call SAS_XVPUT. If you specify the actual location in the calls to SAS_XVPUTD, specify NULL for *base-ptr* in the call to SAS_XVPUT. You must use one method for all variables defined by the gather-output process.

If the NAMESTR for a variable is the same as one that describes the input variable, simply use the address of that structure for the *ptr-to-namestr* parameter. If you want to change some of the values in a NAMESTR, you must create a new NAMESTR and make the changes in that copy. Refer to the description of SAS_XVNAMEI for an example of how to do this.

▶ ***Caution*** .... ***You must not change any of the values in the NAMESTR for an input data set.*** ▲

The number of calls to this function should not be greater than the *number-of-vars* argument in the call to SAS_XVPUTI.

Example

A simple use of this routine is illustrated in the example for SAS_XVPUT.

The following example illustrates how you can gather the data for each output observation from several arrays or other scattered storage areas. This ability to write data from noncontiguous storage areas is the reason these routines are called gather-write routines. For example, you can store all the numeric variables from the input data set in one array and all character variables in a separate array. Then you can create the output data set by gathering variables from these arrays as shown in the example.

Each call to SAS_XVPUT in this example automatically gathers the character variables from `chararray` and the numeric variables from `fltarray`. Before setting up the scatter-read process, the example determines the size of these arrays by testing the length of each variable stored in the NAMESTR for the variable. The example then allocates the necessary storage for the numeric array (the number of numeric variables times the size of a `double`) and the character array (the sum of the lengths of each character variable). This example does not add any new variables to the data set or change the description of the variables by altering the NAMESTR values.

```
ptr    chararray;
double *fltarray;
long   charlens;
ptr    infileid,
       outfileid,
       xvgetptr,
       xvputptr;

short  c,
       j,
       n,
       *list,
       numvars = 0,
       nvar;
struct NAMESTR  *storenam;

access fileids

   /* initialize input buffer and get list*/
nvar = SFN(proc.head, 6);
SAS_XVGETI(infileid, nvar, &xvgetptr);
SAS_XVPUTI(outfileid, nvar, &xvputptr);
list = (short *)SFLP(proc.head, 6);

   /* determine the total number of numeric variables */
   /* and the total lengths of character variables    */
for (j = 0; j < nvar; j++) {
   SAS_XVNAME(infileid, list[j], &storenam);
   if (storenam->ntype == 1) numvars += 1;
   else charlens += (storenam->nlng);
}

   /* allocate space for both arrays */
fltarray = (double *)SAS_XMEMEX(numvars * sizeof(double));
chararray = SAS_XMEMEX(charlens);

   /* scatter read to and gather write from two arrays */
for (j = 0, c = 0, n = 0; j < nvar; j++) {
   SAS_XVNAME(infileid, list[j], &storenam);

      /* use two-dimensional chararray for character variables */
   if (storenam->ntype == 2) {
      SAS_XVGETD(xvgetptr, list[j], 0, chararray + c,
                 storenam->nlng, XV_NOFMT);
      SAS_XVPUTD(xvputptr, storenam, 0, chararray + c, storenam->nlng);
      c += storenam->nlng;
   }
```

SAS_XVPUTD *continued*

```
      /* use floating-point array for numeric variables */
   else {
      SAS_XVGETD(xvgetptr, list[j], 0, fltarray + n, sizeof(double),
                 XV_NOFMT);
      SAS_XVPUTD(xvputptr, storenam, 0, fltarray + n, sizeof(double));
      n += 1;
   }
}
SAS_XVGETE(xvgetptr);
SAS_XVPUTE(xvputptr);

   /* process observations */
while (SAS_XBYGET(infileid) != NULL) {
   while (SAS_XBYGET(infileid) != NULL) {
      SAS_XVGET(xvgetptr, NULL);
      further processing
      SAS_XVPUT(xvputptr, NULL);
      SAS_XOADD(outfileid, xvputptr);
   }
}
```

SAS_XVPUTE

Terminate gather-write definition

Declarations

Type	Variable	Use	Description
ptr	*xvput-ptr*	input	control pointer returned by SAS_XVPUTI
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc = SAS_XVPUTE(xvput-ptr);
```

Description

The SAS_XVPUTE routine indicates that the definition of the gather-output process is completed. This routine is required to signal that no more SAS_XVPUTD calls will be issued. You must call SAS_XVPUTE before you begin calling SAS_XVPUT.

This routine writes all of the NAMESTRs and file-header information that was built by SAS_XVPUTD calls.

Example

Refer to the example for SAS_XVPUTD for an illustration of how to use SAS_XVPUTE.

SAS_XVPUTI

Initialize gather-write operation

Declarations

Type	Variable	Use	Description
ptr	*file-ptr*	input	fileid of an open output data set
int	*number-of-vars*	input	number of variables involved in the gather-write definition
ptr	**ptr-to-xvput-ptr*	output	pointer to the control pointer for all other SAS_XVPUT*x* calls
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc = SAS_XVPUTI(file-ptr, number-of-vars, 0, ptr-to-xvput-ptr);
```

Description

The SAS_XVPUTI routine initializes the process that writes observations to a SAS data set. Before you can use any other SAS_XVPUT*x* routine, you must initialize this process. Be sure that you remember to include all the variables that will be gathered in the gather-write process when you specify the *number-of-vars* argument. For example, if you are gathering variables from the VAR list or the BY list, and you are also creating new variables in your program, you must add together the number of variables in both lists as well as the new variables.

After the call to SAS_XVPUTI, you then use SAS_XVPUTD or SAS_XVPUTCV to establish where to find the variables that are to be moved into the output buffer. When you call SAS_XVPUT to write an observation, the values

SAS_XVPUTI *continued*

for the observation are gathered into the buffer. To actually write the observation to the SAS data set, call SAS_XOADD.

The *number-of-vars* parameter must be at least as large as the number of SAS_XVPUTD calls that follow plus the number of variables processed by calls to SAS_XVPUTCV.

Example

This example illustrates calling SAS_XVPUTI to set up the output data set so it has the number of variables from the input list plus a new variable created by the procedure. Note that you must create a new copy of the NAMESTR to fill in the values for the new variable.

```
double          *observat,
                sum;
ptr             infile,
                outfile,
                out_ptr;
short           *list,
                nvar;
int             x;
struct STLIST   *varlist;
struct NAMESTR newnam,
                *newnam_ptr;

   /* access fileid and variable list */
varlist = (struct STLIST *)SFLD(proc.head, 6);
SAS_XVDFLST(infile, 1, list, 0);
nvar = SFN(proc.head, 6);
list = (short *)SFLP(proc.head, 6);

set up scatter reads

   /* set up gather writes */
SAS_XVPUTI(outfile, nvar+1, &out_ptr);
for (x = 0; x < nvar; x++) {
   SAS_XVNAME(infile, list[x], &newnam_ptr);
   SAS_XVPUTD(out_ptr, newnam_ptr, 0, observat+x, sizeof(double));
}

newnam_ptr = &newnam;

   /* initialize the NAMESTR for the new variable  */
SAS_XVNAMEI(newnam_ptr);
```

```
   /* set NAMESTR information */
newnam.ntype = 1;
newnam.nlng = sizeof(double);
newnam.nname = "SUM";
newnam.namelen = 3;
newnam.nlabel = "Sum of multiplied values";
newnam.nlablen = 24;

   /* add new variable to gather-write process */
SAS_XVPUTD(out_ptr, newnam_ptr, 0, &sum, sizeof(double));

   /* complete gather-write definition */
SAS_XVPUTE(out_ptr);
```

SAS_XVPUTT

Terminate gather write

Declarations

Type	Variable	Use	Description
ptr	*xvput-ptr*	input	control pointer returned by SAS_XVPUTI
rctype	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc = SAS_XVPUTT(xvput-ptr);
```

Description

The SAS_XVPUTT routine frees the memory allocated by the SAS_XVPUTI routine. If you need to free this memory, you must use SAS_XVPUTT; do not use SAS_XMEMFRE.

Note: All memory allocated for your procedure is freed when the procedure finishes executing; therefore, in most cases it is not necessary to free memory allocated by SAS_XVPUTI. If you want to free this memory while your procedure is still executing, you can call SAS_XVPUTT after all SAS_XVPUT calls are completed.

Example

This example illustrates freeing memory allocated for scatter reading and gather writing, as well as the memory allocated to store an observation. Note that memory allocated with SAS_XVGETI and SAS_XVPUTI must be freed with

SAS_XVPUTT *continued*

SAS_XVGETT and SAS_XVPUTT, respectively, while the memory allocated with SAS_XMEMEX must be freed with SAS_XMEMFRE. In all cases, once you have freed the memory, you cannot use the pointer to it again. It is a good idea to set these pointers to NULL.

Note: You can call SAS_XVPUTT only after you have completed all calls to other SAS_XVPUTx routines.

```
double    *observat;
ptr       infile,
          outfile,
          ptr_xvget,
          ptr_xvput;
short     nvar;

   /* allocate memory for storing observation */
observat = (double *)SAS_XMEMEX(nvar * sizeof(double));

   /* define scatter reads and gather writes with specific locations */
SAS_XVGETI(infile, nvar, &ptr_xvget);
SAS_XVPUTI(outfile, nvar, &ptr_xvput);

set up scatter reads and gather writes

   /* read observations */
while (SAS_XBYNEXT(infile) > 0) {
   while (SAS_XBYGET(infile) != NULL) {
      SAS_XVGET(ptr_xvget, NULL);
      process observation
      SAS_XVPUT(ptr_xvput, NULL);
      SAS_XOADD(outfile, ptr_xvput);
   }
}

   /* free memory when no longer needed */
SAS_XVGETT(ptr_xvget);
SAS_XVPUTT(ptr_xvput);
SAS_XMEMFRE(observat);
ptr_xvget = NULL;
ptr_xvput = NULL;
observat = NULL;

other processing after all observations read and written
```

SAS_XZ Routine: Operating System Interface

SAS_XZOSCMD is the only operating system interface routine in SAS/TOOLKIT software.

SAS_XZOSCMD

Invoke an operating system command

Declarations

Type	Variable	Use	Description
ptr	*command-string-ptr*	input	command to execute
long	**system-rc*	output	return code from command
int	*rc*	returned	return code from SAS_XZOSCMD

Usage

```
rc = SAS_XZOSCMD(command-string-ptr, system-rc);
```

Description

This routine enables you to issue operating system commands from within your procedure. The *command-string* must be a valid operating system command string. Keep in mind that using this routine in your procedure makes your procedure not portable.

Under MVS you can use this routine in two ways:

- You can use SAS_XZOSCMD to issue a TSO command. If you use SAS_XZOSCMD in this way, your procedure cannot be run in batch mode.
- You can prefix the command string specified in the call to SAS_XZOSCMD with the string XEQ:, which enables the application to execute using the Attach interface. If you use this form of the SAS_XZOSCMD routine, your procedure can run in either interactive or batch mode.

SAS_XZOSCMD *continued*

Example

The following examples illustrate how to use the SAS_XZOSCMD routine under CMS, MVS, and VMS.

This example illustrates how to list the current disks from your procedure under CMS.

```
int   rc;
long  sysrc;

rc = SAS_XZOSCMD("Q DISK", &sysrc);
```

This example illustrates how to execute the ABC program from your procedure under MVS using the Attach interface.

```
int   rc;
long  sysrc;

rc = SAS_XZOSCMD("XEQ:ABC", &sysrc);
```

This example illustrates how to list the current directory from your procedure under VMS.

```
int   rc;
long  sysrc;

rc = SAS_XZOSCMD("SHDEF", &sysrc);
```

Chapter 23 SAS_Z Routines

Introduction

The routines described in this chapter provide a means of performing commonly needed tasks that require special programming techniques to interface correctly with the SAS System. Some of the sublib routines replace common functions in the language you are using to write your procedure; for example, SAS_ZMOVEI and SAS_ZSTRPOS replace language functions that move strings and locate characters in a string. Other SAS_Z routines, such as the various date routines (SAS_ZDATJUL, SAS_ZJULDAT, SAS_ZJULYMD, SAS_ZTODAY, SAS_ZYMDJUL) and the missing value routines (the SAS_ZMISS series), simply provide special functions that are necessary to write programs that conform to the style of SAS procedures.

Tables 23.1 through 23.10 group the SAS_Z routines by their uses. Note that a SAS_Z routine may appear in more than one table if the function of the routine applies to several catagories.

Routines for Handling Strings

The string-handling routines allow easier manipulation of character strings. Many of these routines are not necessary in the PL/I environment due to its rich array of string manipulation routines. These routines are much more useful in the FORTRAN environment, which has CHARACTER variables but few routines to manipulate character data. The C environment has a small subset of string-handling routines, but they are based most often on null termination. Except as noted in the description of the routine, the SAS_Z routines for character manipulation do not require null terminators for strings.

Several of the string-handling routines validate or build SAS names. Keep in mind the following rules for SAS names:

- Names can be up to eight characters long.
- The first character must be a letter (A, B, C, . . ., Z) or underscore (_). Subsequent characters can be letters, numeric digits (0, 1, . . ., 9), or underscores.
- Embedded blanks cannot appear in SAS names.
- Special characters, except for the underscore, are not allowed.

These routines are very useful in procedures to build variable names for output data sets, or to ensure that variables coming from external databases are properly named as SAS variables.

Table 23.1 summarizes the SAS_Z routines for handling character strings.

Table 23.1 *String-Handling Routines*

Routine	Description
SAS_ZCOMOFF	compares areas
SAS_ZCOMPAR	compares long strings
SAS_ZCOMPUP	compares two strings (non-case sensitive)
SAS_ZNAMFIX	makes a SAS name from a root and numeric suffix
SAS_ZNAMSUF	divides a SAS name into a root and numeric suffix
SAS_ZNAMVER	determines whether a string is a legal SAS name

(continued)

Table 23.1
(continued)

Routine	Description
SAS_ZNAME	converts any 8-byte character string to a valid SAS name
SAS_ZSTOL	scans a character string for an integer and converts it
SAS_ZSTRANC	substitutes one character for another
SAS_ZSTRANS	converts characters in a string
SAS_ZSTRCOM	compares two strings with the shorter string padded with blanks
SAS_ZSTRDEL	deletes characters from a string
SAS_ZSTRIP	returns a string length, omitting trailing blanks
SAS_ZSTRIPQ	removes quotes from a null-terminated string
SAS_ZSTRJLS	performs justification on strings
SAS_ZSTRLEN	returns the length of a null-terminated string
SAS_ZSTRLO	converts a string to lowercase in place
SAS_ZSTRMOV	moves a string with blank padding or truncation
SAS_ZSTRNDX	searches for one string inside another
SAS_ZSTRNOT	searches for a character not equal to a given character
SAS_ZSTRPOS	searches for a character in a string
SAS_ZSTRUP	converts a string to uppercase in place
SAS_ZSTRVER	searches a string for any characters not in a second string

Routines for Converting Data

Data conversion routines enable you to change the following:

- data types, from character to integer and the reverse.
- the case of character data (from uppercase to lowercase and the reverse).
- the storage format (from transport format to local host format and the reverse). The transport routines are especially useful if you want to generate external data files that are to be transported from one operating system to another. If you are generating SAS data sets in transport format, specify the XPORT engine in the LIBNAME statement in the SAS step that creates the data set.

  There are transport formats for integers, floating-point numbers, and characters. Transport formats are the same under all supported operating systems. For Release 6.06, integer transport format corresponds to the IBM mainframe representation, which is unlike the native integer representation under the VMS environment. Floating-point transport format also corresponds to the IBM mainframe representation of floating-point numbers. The character transport format is ASCII. The advantage of using transport format is that you can read your file on any host, and ensure the proper conversion from transport to local format using these routines. Likewise, you can generate transport format fields on any host using these routines.

- the internal representation of data so they can be sorted correctly (called *collatable* format) and the reverse. You use these routines to compare two fields as character data fields. You cannot compare two memory locations as character strings if the memory area contains numeric values, because numbers are not collatable as character strings. For example, the integer value 1 is stored on the IBM mainframe in 2 bytes as '0001'x. The value −1 is stored as 'FFFF'x. The value 'FFFF'x collates higher than '0001'x, but −1 is less than 1. Using the collating routines allows you to convert integers or doubles into fields that compare as characters. Using these routines overcomes the problem described in the previous example. These routines are especially useful if you are building a key record where there can be multiple keys, or if the code that performs the key compare does not know the key types.
  Note that you cannot perform numeric operations on data in collatable format. You must reconvert the data to standard numeric format first.

Table 23.2 summarizes the SAS_Z routines for converting data.

Table 23.2 *Data Conversion Routines*

Routine	Description
SAS_ZCSDBL	converts a float value into a value that can be sorted
SAS_ZCSINT	converts an unsigned integer value into a value that can be sorted
SAS_ZCTCHR	converts a character string into transport format
SAS_ZCTDBL	converts a double into transport format
SAS_ZCTINT	converts an integer into transport format
SAS_ZLTOS	converts a long value into a character string
SAS_ZRSDBL	restores a float value from a value that can be sorted
SAS_ZRSINT	restores an integer from a value that can be sorted
SAS_ZRTCHR	restores a transport character string to local format
SAS_ZRTDBL	restores a transport double to local format
SAS_ZRTINT	restores a transport format integer to local format
SAS_ZSTOL	scans a character string for an integer and converts it
SAS_ZSTRLO	converts a string into lowercase in place
SAS_ZSTRUP	converts string into uppercase in place

Routines for Handling Dates and Times

The date and time routines enable you to

□ retrieve the current system date and time of day in varying formats. You can then use SAS_XFNAME and SAS_XFXPN to format the dates for printing.

□ convert SAS dates, SAS times, and SAS datetime values to other formats and the reverse.

In the SAS System every date is a unique number on a number line. Each SAS date represents the number of days between January 1, 1960, which is 0 on the number line, and the specified date. Dates before January 1, 1960, are negative numbers; those after January 1, 1960, are positive. Valid SAS dates must be in the range of 01JAN1582 A.D to 31DEC20000 A.D.

Note: The SAS System accepts two-digit, four-digit, or five-digit (for dates after 9999) year values. Two-digit year values are attributed to the century specified in the YEARCUTOFF= SAS system option. The current default value for this option is 1900.

A SAS time value is a representation of a time of day in terms of the number of seconds since midnight of the current day. A SAS time value can range from 0 to 86399 (there are 86,400 seconds in a day). SAS time values are independent of SAS dates.

A SAS datetime value is a representation of a time of day for a given date in terms of the number of seconds since January 1, 1960. A SAS datetime value can range from 01JAN1582:00:00 to 31DEC20000:23:59:59.

Table 23.3 summarizes the SAS_Z routines for handling dates and times.

Table 23.3 *Dates and Times Routines*

Routine	Description
SAS_ZDATE	retrieves the current month, day, and year
SAS_ZDATJUL	converts the Julian date into a SAS date
SAS_ZDATPRT	obtains the SAS date portion of a datetime value
SAS_ZDATTIM	obtains the current datetime value
SAS_ZJULDAT	converts the SAS date into the Julian date
SAS_ZJULYMD	converts the year, month, and day into the Julian date
SAS_ZTIME	converts the host computer's specific time to the SAS time value
SAS_ZTODAY	obtains today's date as a SAS date value
SAS_ZYMDJUL	converts the Julian date into year, month, and day

Routines for Filling Memory

These routines provide a shortcut technique for filling an area of memory with a specified value. These routines are also useful when you want to reuse memory and reset it to 0.

Table 23.4 summarizes the SAS_Z routines for filling memory.

Table 23.4
Memory-Filling Routines

Routine	Description
SAS_ZFILLCI	fills the area with a specified character
SAS_ZFILLDI	fills a vector of doubles with a specified value
SAS_ZFILLOI	fills memory with arbitrary objects
SAS_ZZEROI	fills memory with zeros

Routines for Handling Matrices

Table 23.5 summarizes the SAS_Z routines for handling matrices. Use these routines to perform vector or matrix computations on data.

Table 23.5
Matrix-Handling Routines

Routine	Description
SAS_ZISUB	indexes the location of an element in a compressed storage of a symmetric matrix
SAS_ZSCP	accumulates the crossproduct of multiples of rank 1 matrices
SAS_ZSWEEP	sweeps a symmetric matrix on a given pivot
SAS_ZVDOTJJ	computes the dot product of two vectors of possibly different strides
SAS_ZVXAYJJ	replaces a vector with the sum of the vector and the multiple of another vector of possibly different strides
SAS_ZVZJJJ	computes the sum of the vector and the multiple of another vector of possibly different strides and stores the result in another vector

Routines for Mathematics and Statistics

If you are developing a procedure that performs special mathematical or statistical calculations, you may want to use these routines.

Table 23.6 summarizes the SAS_Z routines for mathematics and statistics.

Table 23.6
Mathematics and Statistics Routines

Routine	Description
SAS_ZLGAMMA	computes the natural logarithm of the Gamma function
SAS_ZPGT2	computes the probability that the absolute value of a random variable from the Student's *t*-distribution is greater than or equal to *t*
SAS_ZPGX2	computes the probability that the value of a random variable from the chi-square distribution is greater than or equal to x2
SAS_ZPRBF	computes the probability that the value of a random variable from the *F*-distribution is greater than or equal to *f*
SAS_ZPRBFMT	formats a probability value
SAS_ZPRBNRM	computes the probability from the normal distribution
SAS_ZPROBIT	computes the quantile from the normal distribution
SAS_ZPSIG	determines the format for printing a number with a given number of significant digits
SAS_ZPSIG2	determines the format for printing two numbers so that each has a given number of significant digits
SAS_ZQC	computes the quantile from the chi-square distribution
SAS_ZQF	computes the quantile from the *F*-distribution
SAS_ZQT	computes the quantile from the Student's *t*-distribution
SAS_ZRANUNI	generates a random variate from a uniform distribution

Routines for Handling Missing Values

If a numeric variable has no value, the SAS System stores a special floating-point value to indicate that the variable has no value. For example, the result of a divide by zero is a missing value. A missing value also occurs if the data are not present at all.

The SAS System permits the following 28 different missing values for numeric variables:

- □ the standard missing value, indicated by a single dot (.)
- □ special missing values, corresponding to each letter of the alphabet (.A, .B, . . ., .Z)
- □ the special missing value for the underscore, ._.

These routines allow you to check for or create missing values.

▶ *Caution* .... ***Do not attempt to determine if a numeric value is missing by any other means than with these routines. In addition, do not attempt any numeric calculations with missing values.*** ▲

You cannot perform numeric calculations on missing values because of the way some operating systems store missing values. Numeric operations on missing values may cause premature termination of the SAS job.

Differences in how missing values are stored on different hosts is also the reason why you cannot check for a missing value using standard comparison operators. For example, on the IBM mainframe, a missing value is treated the same as a 0. However, if you compare a missing value and a non-missing value (using standard comparison operators) on the IBM mainframes, the result always indicates equality. You must use the SAS_Z routines for missing values to obtain proper results.

Table 23.7 summarizes the SAS_Z routines for handling missing values.

Table 23.7 *Missing Value Routines*

Routine	Description
SAS_ZMISS	tests whether a `double` is a missing value
SAS_ZMISSN	determines if any elements of an array are missing
SAS_ZMISSV	returns a SAS missing value

Routines for Moving Data

These routines provide shortcut techniques for moving data. Note that these routines can replace some built-in functions of the programming language you are using, or they may provide functions not included in a language.

Table 23.8 summarizes the SAS_Z routines for moving data.

Table 23.8 *Data Movement Routines*

Routine	Description
SAS_ZMOVEI	copies a block of memory
SAS_ZNDMOVI	moves memory nondestructively
SAS_ZSTRMOV	moves a string with blank padding or truncation

Routines for Formatting Data

These routines perform numeric formatting. The SAS_ZLTOS and SAS_ZSTOL are shortcut routines that you can use instead of SAS_XFXPN and SAS_XFXIN if you know the numeric values are integers. The padding and truncation routines are useful if you need to store numbers in truncated form and then restore them to doubles before using them.

Note: Do not use truncated values in numeric computations. Note also that truncated data cannot be converted to transport format to move them to other operating systems.

Table 23.9 summarizes the SAS_Z routines for formatting data.

Table 23.9
Data-Formatting Routines

Routine	Description
SAS_ZFPAD	pads a `float` to full double precision
SAS_ZFTRNC	truncates a double-precision value to a `double` of specified length
SAS_ZLTOS	converts a `long` value into a character string
SAS_ZPFMT	determines the number of decimal places for printing
SAS_ZPRBFMT	formats a *p*-value
SAS_ZPSIG	determines the format for printing a number with a given number of significant digits
SAS_ZPSIG2	determines the format for printing two numbers so each has a given number of significant digits
SAS_ZSTOL	scans a character string for an integer and converts it

Miscellaneous Routines

These routines perform several miscellaneous functions:

- accessing character list values in the statement structure
- accessing information about the current host environment
- building and maintaining balanced binary trees. These routines are especially useful with a program that reads values that may not be sorted and finds or inserts the value in a sorted order.

Table 23.10 lists the miscellaneous SAS_Z routines.

Table 23.10
Miscellaneous Routines

Routine	Description
SAS_ZPARM	returns a pointer to the *n*th element of a list of extra long character strings (mode 5 of the @LIST semantic)
SAS_ZPARMN	returns the number of elements in a list of extra long character strings (mode 5 of the @LIST semantic)
SAS_ZSYINF	returns a description of the host operating system
SAS_ZTREBLD	locates or adds a node to a binary tree
SAS_ZTREFND	searches a binary tree
SAS_ZTRELST	converts a tree built by SAS_ZTREBLD into a linked list

Description of SAS_Z Routines

The SAS_Z routines are described in alphabetical order in the remainder of this chapter.

Note: Many of the SAS_Z routines are illustrated in this chapter. For other examples of how to use the SAS_Z routines, refer to the EXAMPLE procedure and the SUBLIB procedure, included on the installation tape sent to your site. Check with your SAS Software Representative to locate the source code for PROC EXAMPLE and PROC SUBLIB.

SAS_ZCOMOFF

Compare areas

Declarations

Type	Variable	Use	Description
ptr	*area-1-ptr*	input	pointer to area 1.
ptr	*area-2-ptr*	input	pointer to area 2.
int	*type-length*	input	the type and length of the fields to compare: >0 = character, with this length 0 = integer −2 = double floating-point.
int	*offset*	input	offset to the key field. This value cannot be greater than 4096.
int	*result*	returned	indicates the result of the comparison: >0 = the value pointed to by *area-1-ptr* is greater than the value pointed to by *area-2-ptr* 0 = both areas are the same <0 = the value pointed to by *area-1-ptr* is less than the value pointed to by *area-2-ptr*.

Usage

```
result = SAS_ZCOMOFF(area-1-ptr, area-2-ptr, type-length, offset);
```

Description

The SAS_ZCOMOFF routine compares two values of the same type. If the areas to be compared are character values, they must be declared as `unsigned char` values (on systems that accept that declaration). Data values should be correctly aligned. If you compare double floating-point values, a missing value is valid input. If both values are missing, they are compared in their relative order.

SAS_ZCOMOFF *continued*

Example

The following example illustrates using SAS_ZCOMOFF for various types of data:

```
int     i,
        ivalue1,
        ivalue2;
double  value1,
        value2;

   /* SAS_ZCOMOFF for characters */
i = SAS_ZCOMOFF("ABC", "CBA", 3, 0);
SAS_XPSLOG("SAS_ZCOMOFF for ABC/CBA should be negative: %d.", i);

   /* SAS_ZCOMOFF for doubles */
value1 = value2 = 1;
i = SAS_ZCOMOFF((ptr)&value1, (ptr)&value2, -2, 0);
SAS_XPSLOG("SAS_ZCOMOFF for 1 and 1 should be zero: %d.", i);

   /* SAS_ZCOMOFF for integers */
ivalue2 = -1;
i = SAS_ZCOMOFF((ptr)&ivalue1, (ptr)&ivalue2, 0, 0);
SAS_XPSLOG("SAS_ZCOMOFF for 0 and -1 should be positive: %d.", i);
```

SAS_ZCOMPAR

Compare long strings

Declarations

Type	Variable	Use	Description
ptr	*area-1-ptr*	input	pointer to area of bytes to be compared.
ptr	*area-2-ptr*	input	second area to compare. If this pointer is NULL, the data pointed to by *area-1-ptr* are compared to blanks.
int	*length*	input	the number of bytes to compare. If this value is 0, *result* is set to 0.
int	*result*	returned	result of comparison: >0 = *area-1* is greater than *area-2* 0 = the areas are the same <0 = *area-2* is greater than *area-1*.

Usage

```
result = SAS_ZCOMPAR(area-1-ptr, area-2-ptr, length);
```

Description

The SAS_ZCOMPAR routine compares two character strings that have the same length and indicates which is greater.

Example

The following example compares several character strings:

```
SAS_XPSLOG("SAS_ZCOMPAR should compare as 0: %d.",
           SAS_ZCOMPAR("ABC", "ABC", 3));
SAS_XPSLOG("SAS_ZCOMPAR should compare as negative: %d.",
           SAS_ZCOMPAR("ABC", "XYZ", 3));
SAS_XPSLOG("SAS_ZCOMPAR should compare as positive: %d.",
           SAS_ZCOMPAR("XYZ", "ABC", 3));
```

SAS_ZCOMPUP

Compare two strings (non-case sensitive)

Declarations

Type	Variable	Use	Description
ptr	*area-1-ptr*	input	pointer to area of bytes to be compared.
ptr	*area-2-ptr*	input	second area to compare. If this pointer is NULL, the data pointed to by *area-1-ptr* are compared to blanks.
int	*length*	input	length of area (in bytes) to compare. If this value is 0, *result* is set to 0.
int	*result*	returned	indicator of sort order: >0 = the string pointed to by *area-2-ptr* sorts first 0 = the two strings are equal <0 = the string pointed to by *area-1-ptr* sorts first.

Usage

```
result = SAS_ZCOMPUP(area-1-ptr, area-2-ptr, length);
```

SAS_ZCOMPUP *continued*

Description

The SAS_ZCOMPUP routine compares two character strings as if both strings were uppercase characters and indicates which is greater. That is, SAS_ZCOMPUP compares strings without being case sensitive to determine which string appears first in a sorted sequence.

Example

The following example compares several mixed-case character strings:

```
SAS_XPSLOG("SAS_ZCOMPUP should compare as 0: %d.",
           SAS_ZCOMPUP("ABC", "abc", 3));
SAS_XPSLOG("SAS_ZCOMPUP should compare as negative: %d.",
           SAS_ZCOMPUP("ABC", "xyz", 3));
SAS_XPSLOG("SAS_ZCOMPUP should compare as positive: %d.",
           SAS_ZCOMPUP("XYZ", "abc", 3));
```

SAS_ZCSDBL

Convert a float value into a value that can be sorted

Declarations

Type	Variable	Use	Description
double or float	*value*	update	the floating-point value to be converted.
int	*length*	input	number of bytes for storing the float value. The value must be in the range of 2 to 8.
long	*rc*	returned	return code: 0 = successful.

Usage

C	`rc = SAS_ZCSDBL((ptr) &value, length);`
FORTRAN	`rc = SAS_ZCSDBL(IADDR(value), length)`
PL/I	`rc = SAS_ZCSDBL(ADDR(value), length);`

Description

The SAS_ZCSDBL routine creates a coded form of a floating-point number so it can be sorted as if it were a character variable. The conversion is performed in place; that is, the value pointed to by *value-ptr* is changed. To reverse the coded value, use the SAS_ZRSDBL routine. If you want to group several values together to compare and sort by more than one value, use SAS_ZCSDBL for each value. All numeric values are converted to character strings in the proper ascending or descending order.

Example

The following example converts a double-precision floating-point value to a character that can be sorted:

```
double  value;
char    temp[200],
        temp2[200];
int     i;

value = -1;
SAS_ZMOVEI(&value, temp, sizeof(double));
SAS_ZCSDBL(temp, sizeof(double));
value = 1;
SAS_ZMOVEI(&value, temp2, sizeof(double));
SAS_ZCSDBL(temp2, sizeof(double));

   /* test comparison of the collatable values */
i = SAS_ZCOMPAR(temp, temp2, sizeof(double));
SAS_XPSLOG("SAS_ZCOMPAR should return negative for comparison of
collatable doubles -1 and 1: %d.", i);
```

SAS_ZCSINT

Convert an unsigned integer value into a value that can be sorted

Declarations

Type	Variable	Use	Description
unsigned int, unsigned short, or unsigned long	*int-value*	update	the unsigned integer to be converted

int	*length*	input	length in bytes of the integer value. This value should be one of the following:
			`sizeof(int)`
			`sizeof(long)`
			`sizeof(short)`.

long	*rc*	returned	return code: 0 = successful.

SAS_ZCSINT *continued*

Usage

C	`rc = SAS_ZCSINT((ptr) &int-value, length);`
FORTRAN	`rc = SAS_ZCSINT(IADDR(int-value), length)`
PL/I	`rc = SAS_ZCSINT(ADDR(int-value), length);`

Description

The SAS_ZCSINT routine creates a coded form of an integer so it can be sorted as if it were a character variable. The conversion is performed in place; that is, the *int-value* is changed. To reverse the coded value, use the SAS_ZRSINT routine.

Example

See the example for the SAS_ZRSINT routine.

SAS_ZCTCHR

Convert a character string to transport format

Declarations

Type	Variable	Use	Description
ptr	*source-ptr*	input	address of string to be converted
int	*length*	input	length in bytes of string
ptr	*destination-ptr*	update	address of buffer for transport data
long	*rc*	returned	return code: 0=successful

Usage

```
rc = SAS_ZCTCHR(source-ptr, length, destination-ptr);
```

Description

The SAS_ZCTCHR routine converts a character string to transport format, which is the extended ASCII representation. All 8 bits of each byte are significant. After you have converted a value to transport format, you cannot change the value. You can only store it or move it.

Example

The following example converts a character string to transport format, prints the character string using the $HEX format, and then converts the data in transport format back to character data:

```
ptr    temp,
       temp2;
long   fcode;

SAS_XFNAME("$HEX    ", 'F', NULL, &fcode);
SAS_ZCTCHR("ABC", 3, temp);
SAS_XFXPC(temp, 3, fcode, temp2, 6);
SAS_XPSLOG("Hex representation of 'ABC' in transport format: %6s
(should be 414243).", temp2);
SAS_ZRTCHR(temp, 3, temp2);
SAS_XPSLOG("Value converted back from transport '%3s' (should be ABC).",
        temp2);
```

SAS_ZCTDBL

Convert a double to transport format

Declarations

Type	Variable	Use	Description
double	*source*	input	double to convert to transport format
ptr	*destination-ptr*	input	address of buffer for transport double
int	*out-length*	input	length of transport double (2-8)
long	*rc*	returned	return code: 0 = successful nonzero = unsuccessful

Usage

```
rc = SAS_ZCTDBL(source, destination-ptr, out-length);
```

Description

The SAS_ZCTDBL routine converts a double to transport format. After you have converted a value to transport format, you cannot change the value. You can only store it or move it.

SAS_ZCTDBL *continued*

Example

The following example converts a double-precision floating-point value to transport format, prints the data in transport format using the $HEX format, and then converts the data in transport format back to a double-precision floating-point value:

```
ptr    temp,
       temp2;
long   fcode;

SAS_XFNAME("$HEX    ", 'F', NULL, &fcode);
SAS_ZCTDBL(1.0, temp, 8);
SAS_XFXPC(temp, 8, fcode, temp2, 16);
SAS_XPSLOG("Hex representation of (double)1 in transport format: %16s
(should be 4110000000000000).", temp2);
SAS_XPSLOG("Value converted back from transport: %f (should be 1.00)",
           SAS_ZRTDBL(temp, 8));
```

SAS_ZCTINT

Convert an integer to transport format

Declarations

Type	Variable	Use	Description
int, long, short	*source*	input	integer to convert to transport format
ptr	*destination-ptr*	input	address of buffer for transport integer
int	*out-length*	input	length of transport integer (2 or 4)
long	*rc*	returned	return code: 0=successful

Usage

C	`rc= SAS_ZCTINT((long)source, destination-ptr, out-length);`
FORTRAN	`rc= SAS_ZCTINT(source, destination-ptr, out-length)`
PL/I	`rc= SAS_ZCTINT(source, destination-ptr, out-length);`

Description

The SAS_ZCTINT routine converts an integer, short or long, to transport format. After you have converted a value to transport format, you cannot change the value. You can only store it or move it.

Example

The following example converts an integer value to transport format, prints the data in transport format using the $HEX format, and then converts the data in transport format back to an integer value:

```
ptr    temp,
       temp2;
long   fcode;

SAS_XFNAME("$HEX    ", 'F', NULL, &fcode);
SAS_ZCTINT(10, temp, 4);
SAS_XFXPC(temp, 4, fcode, temp2, 8);
SAS_XPSLOG("Hex representation of (long)10 in transport format: %8s
(should be 0000000A).", temp2);
SAS_XPSLOG("Value converted back from transport: %ld (should be 10)",
           SAS_ZRTINT(temp, 4));
```

SAS_ZDATE

Retrieve the current month, day, and year

Declarations

Type	Variable	Use	Description
int	**month*	output	returned value contains the current month (1—12)
int	**day*	output	returned value contains the current day (1—31)
int	**year*	output	returned value contains the current year (4-digit year)
long	*rc*	returned	return code: 0 = successful

Usage

```
rc = SAS_ZDATE(month, day, year);
```

Description

The SAS_ZDATE routine returns the current month, day, and year.

SAS_ZDATE *continued*

Example

See the example for the SAS_ZJULYMD routine.

SAS_ZDATJUL

Convert Julian date to a SAS date

Declarations

Type	Variable	Use	Description
double	*julian*	input	Julian date. Noninteger values are truncated to integers.
double	*SAS-date*	returned	unformatted SAS date value: missing if *julian* is missing or invalid.

Usage

```
SAS-date = SAS_ZDATJUL(julian);
```

Description

The SAS_ZDATJUL converts Julian dates into SAS date values. The Julian date must be within the range of 01JAN1582 through 31DEC20000. The Julian date can include a four-digit year (for example, 1990001 for 01JAN1990) for dates prior to 10000 A.D., and a five-digit year for dates between 01JAN10000 to 31DEC20000, or it can specify a two-digit year.

Note: If a two-digit year is specified, the century for that year is determined by the value of the YEARCUTOFF= SAS system option. The current default value for this option is 1900.

Example

The following example illustrates how to convert a SAS date value, −1749, to a Julian date. The example then converts the returned Julian date to a SAS data value. The example also prints the dates.

```
double  temp;

temp = SAS_ZJULDAT(-1749.0);
SAS_XPSLOG("19MAR1955 as Julian date should be 1955078: %ld.",
           (long)temp);
temp = SAS_ZDATJUL(temp);
SAS_XPSLOG("SAS_ZDATJUL should return -1749: %ld.", (long)temp);
```

SAS_ZDATPRT

Obtain the SAS date portion of a datetime value

Declarations

Type	Variable	Use	Description
double	*SAS-date-time*	input	SAS datetime value
double	*SAS-date*	returned	unformatted SAS date value

Usage

```
SAS-date = SAS_ZDATPRT(SAS-date-time);
```

Description

The SAS_ZDATPRT extracts the SAS date value from a SAS datetime value.

Example

The following example illustrates how to obtain the current date and time and then extract the date portion from the datetime value. The example also prints the date using the DATE9. format.

```
double  temp;
long    fcode;
char    string[200];

temp = SAS_ZDATTIM();
temp = SAS_ZDATPRT(temp);
SAS_XFNAME("DATE    ", 'F', NULL, &fcode);
SAS_XFXPN(temp, 9, 0, fcode, string);
SAS_XPSLOG("Date part of this %ld (%9s).", (long)temp, string);
```

SAS_ZDATTIM

Obtain the current datetime value

Declarations

Type	Variable	Use	Description
double	*SAS-date-time*	returned	current SAS datetime value

SAS_ZDATTIM *continued*

Usage

```
SAS-date-time = SAS_ZDATTIM();
```

Description

The SAS_ZDATTIM routine returns the current SAS datetime value.

Example

The following example illustrates how to obtain the current date and time as a SAS datetime value and print this value using the DATETIME16. SAS format:

```
double  temp;
long    fcode;
char    string[200];

temp = SAS_ZDATTIM();
SAS_XFNAME("DATETIME", 'F', NULL, &fcode);
SAS_XFXPN(temp, 16, 0, fcode, string);
SAS_XPSLOG("Current datetime is %ld (%16s).", (long)temp, string);
```

SAS_ZFILLCI

Fill area with specified character

Declarations

Type	Variable	Use	Description
char	*fill-char*	input	the character to fill the area with, enclosed in single quotes
ptr	*area-ptr*	output	pointer to the area to be filled
int	*length*	input	length of the area to fill

Usage

C	`SAS_ZFILLCI(`*fill-char, area-ptr, length*`);`
FORTRAN	`CALL SAS_ZFILLCI(`*fill-char, area-ptr, length*`)`
PL/I	`CALL SAS_ZFILLCI(`*fill-char, area-ptr, length*`);`

Description

The SAS_ZFILLCI routine fills an area with a specific character.

SAS_ZFILLDI

Fill a vector of doubles with a specified value

Declarations

Type	Variable	Use	Description
double	*value*	input	the double to fill the vector with.
double	**vector-ptr*	output	pointer to vector to be filled. You must allocate the vector and set the pointer before the call. SAS_ZFILLDI then fills the vector with the specified value.
int	*number*	input	number of doubles in area to fill.

Usage

C	`SAS_ZFILLDI(value, vector-ptr, number);`
FORTRAN	`CALL SAS_ZFILLDI(value, vector-ptr, number)`
PL/I	`CALL SAS_ZFILLDI(value, vector-ptr, number);`

Description

The SAS_ZFILLDI routine fills an array of doubles with a specific value.

SAS_ZFILLOI

Fill memory with arbitrary objects

Declarations

Type	Variable	Use	Description
ptr	*fill-ptr*	input	pointer to the object to fill with.
int	*fill-length*	input	the size of the object. Specify −1 if the string is null-terminated.
ptr	*to-ptr*	output	pointer to the area to fill.
int	*to-number*	input	the number of objects to fill with.

SAS_ZFILLOI *continued*

Usage

C	`SAS_ZFILLOI(fill-ptr, fill-length, to-ptr, to-number);`
FORTRAN	`CALL SAS_ZFILLOI(fill-ptr, fill-length, to-ptr, to-number)`
PL/I	`CALL SAS_ZFILLOI(fill-ptr, fill-length, to-ptr, to-number);`

Description

The SAS_ZFILLOI routine fills an area of memory with the specified value. This routine enables you to initialize arrays of structures.

SAS_ZFPAD

Pad a short float to full double-precision

Declarations

Type	Variable	Use	Description
ptr	*value-ptr*	input	source data to be padded
int	*length*	input	length of source data
double	*number	output	value padded to a full double-precision
long	*rc*	returned	return code:
			0 = successful
			nonzero = unsuccessful

Usage

C	`rc = SAS_ZFPAD(value-ptr, length, &number);`
FORTRAN	`rc = SAS_ZFPAD(value-ptr, length, IADDR(number))`
PL/I	`rc = SAS_ZFPAD(value-ptr, length, ADDR(number));`

Description

The SAS_ZFPAD routine pads short floating-point numbers to double-precision floating-point numbers. This routine ensures that floating-point numbers are padded correctly for the host environment in which they are running.

Example

See the example for the SAS_ZFTRNC routine.

SAS_ZFTRNC

Truncate a double-precision value to a double of specified length

Declarations

Type	Variable	Use	Description
double	*number*	input	value to be truncated
ptr	*value-ptr*	output	pointer to truncated data
int	*length*	input	length of source data
long	*rc*	returned	return code:
			0 = successful

Usage

```
rc = SAS_ZFTRNC(number, value-ptr, length);
```

Description

The SAS_ZFTRNC routine truncates double-precision floating-point numbers to a floating-point number of the specified length. This routine ensures that floating-point numbers are truncated correctly for the host environment in which they are running.

Example

The following example truncates a double-precision floating-point value to a single-precision floating-point value stored in a character string and then restores the value to double-precision:

```
double  value;
char    temp[200];

value = 1;
SAS_ZFTRNC(value, temp, 4);
SAS_ZFPAD(temp, 4, &value);
SAS_XPSLOG("Re-padded double should be 1.00: %f.", value);
```

SAS_ZISUB

Index the location of an element in a compressed storage of a symmetric matrix

Declarations

Type	Variable	Use	Description
int	*i*	input	the actual row index of the element
int	*j*	input	the actual column index of the element
long	*index*	returned	the index in the compressed storage of the matrix

Usage

```
index = SAS_ZISUB(i, j);
```

Description

Compressing a symmetric matrix saves storage space for the matrix. To compress a symmetric matrix, sequentially store in a one-dimensional array the individual rows of the lower part (where $i>=j$) of a symmetric matrix. For example, note the values in this symmetric matrix:

$$\begin{pmatrix} 1.0 & -3.0 & 4.0 \\ -3.0 & 8.0 & 6.0 \\ 4.0 & 6.0 & 9.0 \end{pmatrix}$$

The compressed storage of this symmetric matrix is as follows:

$$\mathbf{A}' = (1.0 \ \ -3.0 \ \ 8.0 \ \ 4.0 \ \ 6.0 \ \ 9.0)$$

To point to an element (i,j) in the matrix **A**, point to the corresponding location in the compressed storage. The SAS_ZISUB routine returns the location, l, for an actual element $A(i,j)$.

In this example, if you want to access the value 8.0, pass the values of 2,2 for i,j, and SAS_ZISUB returns 3, the index of the appropriate element in the compressed storage.

SAS_ZJULDAT

Convert SAS date to Julian date

Declarations

Type	Variable	Use	Description
double	*SAS-date*	input	SAS date value
double	*julian*	returned	Julian date

Usage

```
julian = SAS_ZJULDAT(SAS-date);
```

Description

The SAS_ZJULDAT routine converts SAS date values into Julian dates. The value of *julian* is missing if *SAS-date* is outside the range of 01JAN1582 through 31DEC20000. The Julian date returned specifies a four-digit year (for example, 1990001 for 01JAN1990) for dates prior to 10000 A.D., and a five-digit year for dates between 01JAN10000 to 31DEC20000.

SAS_ZJULYMD

Convert year/month/day to Julian date

Declarations

Type	Variable	Use	Description
double	*year*	input	year value: 0 to 99 = a two-digit year with the century determined by the YEARCUTOFF= SAS system option. 100 to 1581 = invalid values. 1582 to 20000 = valid values. less than 0 or greater than 20000 = invalid values.
double	*month*	input	month value in the range of 1 to 12.
double	*day*	input	day value in the range of 1 to 31. The day must be valid for the month specified.
double	*julian*	returned	the Julian date.

SAS_ZJULYMD *continued*

Usage

```
julian = SAS_ZJULYMD(year, month, day);
```

Description

The SAS_ZJULYMD routine converts year, month, and day values into Julian dates. The value of *julian* is missing if any of the *year*, *month*, or *day* values are missing or if any of these values are invalid.

Example

The following example obtains the current date and stores it in three integers. The example then converts the year, month, and day to a Julian date and prints that date. Finally, the example reconverts the Julian date to separate year, month, and day values.

```
int    month,
       day,
       year;
double temp,
       y,
       m,
       d;

   /* obtain date in three integers */
SAS_ZDATE(&month, &day, &year);
SAS_XPSLOG("Today's month=%d day=%d year=%d.", month, day, year);

y=(double)year;
m=(double)month;
d=(double)day;

   /* convert integers to Julian date */
temp = SAS_ZJULYMD(y, m, d);
SAS_XPSLOG("SAS_ZJULYMD returns : %ld.", (long)temp);

   /* recreate three integers */
SAS_ZYMDJUL(&y, &m, &d, &t);
SAS_XPSLOG("SAS_ZYMDJUL should return same integers: %d/%d/%d.",
        (int)y, (int)m, (int)d);
```

SAS_ZLGAMMA

Compute the natural logarithm of the Gamma function

Declarations

Type	Variable	Use	Description
double	*x*	input	must be a positive number.
double	*y*	returned	the computed logarithm of the Gamma function of *x*. The return value is nonnegative. If x<=0, a value of −99 is returned.

Usage

```
y = SAS_ZLGAMMA(x);
```

Description

The SAS_ZLGAMMA function computes the natural logarithm of the Gamma function. A Rational approximation is used for $x<15$, while an asymptotic expansion is used for $x>=15$.

SAS_ZLTOS

Convert a long value into a character string

Declarations

Type	Variable	Use	Description
long	*value*	input	value to be converted to a character string
char	**string-ptr*	output	address of the character value to be written
int	*length*	input	length in bytes of the converted string
int	*rc*	returned	return code: 0 = successful. nonzero = unsuccessful. Usually the length of the string is too short to store the value.

Usage

```
rc =   SAS_ZLTOS( value , string-ptr, length );
```

SAS_ZLTOS *continued*

Description

The SAS_ZLTOS routine converts the long integer *value* to a character string pointed to by *string-ptr*. The character string is right-justified in the area provided.

Example

The following example converts a long integer to a character string:

```
char  temp[5];
int   i;

i = SAS_ZLTOS(1L,temp,5);
SAS_XPSLOG("SAS_ZLTOS created '%5s' (should be '    1').",temp);
```

SAS_ZMISS

Test whether a double is a missing value

Declarations

Type	Variable	Use	Description
double	*value*	input	value to be tested
int	*miss-code*	returned	indicates whether the value is missing: 0 = a non-missing value was found nonzero = a missing value was found

Usage

```
miss-code = SAS_ZMISS(value);
```

Description

The SAS_ZMISS routine tests a floating-point variable to determine if the variable's value is missing. If *miss-code* is not 0, the value returned is a character indicating the type of missing value found. Typically, SAS_ZMISS is used as follows:

```
if (!SAS_ZMISS(value)) {
   actions for when value is not missing
}
```

Example

The following example sets a missing value using the SAS_ZMISSV routine and then tests the value:

```
double  temp;

temp = SAS_ZMISSV('.');
if (SAS_ZMISS(temp))
   SAS_XPSLOG("The value is missing.");
```

SAS_ZMISSN

Determine if any elements of an array are missing

Declarations

Type	Variable	Use	Description
double	*array[]*	input	array of doubles.
int	*length*	input	length of array. If *length* is 0, 0 is returned for *miss-code*.
int	*miss-code*	returned	indicates if value is missing: 0 = non-missing value nonzero = a missing value was found.

Usage

```
miss-code = SAS_ZMISSN(array, length);
```

Description

The SAS_ZMISSN routine tests an array of floating-point variables to determine if any of the variables' values are missing. If *miss-code* is not 0, the value returned is a character indicating the type of missing value found.

Example

This example sets a missing value in an array using the SAS_ZMISSV routine. The example then uses the SAS_ZMISSN routine to detect that the array contains a missing value.

```
double temp2[3];
int    i;
```

SAS_ZMISSN *continued*

```
temp2[0] = 0;
temp2[1] = SAS_ZMISSV('.');
temp2[2] = 2;
if (SAS_ZMISSN(temp2,3) != 0)
   SAS_XPSLOG("The array contains a missing value.");
```

SAS_ZMISSV

Return a SAS missing value

Declarations

Type	Variable	Use	Description
int	*character*	input	missing value character. For PL/I and FORTRAN languages, declare this variable as a 1-byte character. Acceptable values are '.' 'A' through 'Z' (lowercase characters are uppercased) '_'
double	*missing*	returned	SAS missing value with the specified character.

Usage

```
missing = SAS_ZMISSV(character);
```

Description

The SAS_ZMISSV routine returns a special missing value. If you need to create missing values other than the standard representation (.), pass a character 'A' through 'Z' or '_' and the routine returns the corresponding special missing value, '.A' through '.Z' or '._'. For example, you need to create a special type of missing value in a file that is used to update a SAS data set; existing values can be replaced by missing values only if the missing value has the special code (._). In this case, you should call SAS_ZMISSV with '_' as the *character* argument to produce the value for the update data set.

In addition to using this routine, in C programs you can also use the MACMISSING macro for the standard missing value. All other missing values should be derived by calling SAS_ZMISSV.

Example

See the example for the SAS_ZMISSN routine.

SAS_ZMOVEI

Copy a block of memory

Declarations

Type	Variable	Use	Description
ptr	*from-ptr*	input	pointer to area to move from
ptr	*to-ptr*	output	pointer to area to move to
int	*length*	input	number of bytes to move

Usage

C	`SAS_ZMOVEI(from-ptr, to-ptr, length);`
FORTRAN	`CALL SAS_ZMOVEI(from-ptr, to-ptr, length)`
PL/I	`CALL SAS_ZMOVEI(from-ptr, to-ptr, length);`

Description

The SAS_ZMOVEI routine copies a specified number of bytes from one location to another. This routine differs from SAS_ZNDMOVI, which performs a nondestructive move. Under CMS and MVS, the type of move performed by SAS_ZMOVEI propagates a single value to an entire area of memory. Other operating systems may have other results. If you specifically want to propagate values, use the SAS_ZFILLxI routines, which perform in the same manner under all operating systems.

```
char area[] = "abcdefghijklmn";
char *area_1, *area_2;

area_1 = area;
area_2 = area+3;
SAS_ZMOVEI(area_1, area_2, 5);
```

The contents of *area_1* after this move are as follows:

```
abcabcabijklm
```

SAS_ZNAME

Convert any 8-byte character string to a valid SAS name

Declarations

Type	Variable	Use	Description
char8	*value-ptr*	update	8-byte character string to be converted to a valid SAS name

Usage

C	`SAS_ZNAME(value-ptr);`
FORTRAN	`CALL SAS_ZNAME(value-ptr)`
PL/I	`CALL SAS_ZNAME(value-ptr);`

Description

This routine converts any character string to a valid SAS name by making the following changes:

- replaces lowercase letters with uppercase
- prefixes an underscore to names beginning with a number
- replaces a character that is not a letter or number with an underscore
- converts \0 and trailing characters to blanks
- makes the following changes in numeric strings:

.	is replaced with D
−	is replaced with N
+	is replaced with P.

Example

See the example for the SAS_ZNAMVER routine.

SAS_ZNAMFIX

Make a SAS name from a root and numeric suffix

Declarations

Type	Variable	Use	Description
char	**root-ptr*	input	root value. This value is truncated if necessary to accommodate the suffix.
int	*root-len*	input	length of root or -1 if the value pointed to by *root-ptr* is null-terminated.
long	*suffix*	suffix	value of suffix. The maximum number of digits is 7.
int	*min*	input	minimum number of digits to put in the suffix. Leading 0s are supplied as needed.
char	**name-ptr*	output	returned name formed from root and suffix.
int	*name-len*	output	length of area to receive name. If *name-len* is less than 0, the length is the absolute value of *name-len* and the string pointed to by *name-ptr* is null-terminated.

Usage

C	`SAS_ZNAMFIX(root-ptr, root-len, suffix, min, name-ptr, name-len);`
FORTRAN	`CALL SAS_ZNAMFIX(root-ptr, root-len, suffix, min,name-ptr, name-len)`
PL/I	`CALL SAS_ZNAMFIX(root-ptr, root-len, suffix, min, name-ptr, name-len);`

Description

The SAS_ZNAMFIX routine appends an integer to a name. If the combined length of the name and the suffix is greater than 8, the root portion of the name is truncated to allow the entire suffix.

SAS_ZNAMSUF

Divide a SAS name into root and numeric suffix

Declarations

Type	Variable	Use	Description
char	**name-ptr*	input	name to be separated.
int	*name-len*	input	length of name. Set this to −1 if the string pointed to by *name-ptr* is null-terminated.
char	**root-ptr*	output	pointer to the area where the returned root is stored.
int	*root-len*	input	length of area to receive the root. If *root-len* is less than 0, the length is the absolute value of *root-len* and the string pointed to by *root-ptr* is null-terminated.
long	**suffix-ptr*	output	the value of the suffix. The maximum number of digits is 7.
int	*digits*	returned	number of digits in the suffix.

Usage

```
digits = SAS_ZNAMSUF(name-ptr, name-len, root-ptr, root-len, suffix-ptr);
```

Description

The SAS_ZNAMSUF routine strips the numeric suffix from a SAS name. The root and the suffix are stored in separate fields.

SAS_ZNAMVER

Determine whether a string is a legal SAS name

Declarations

Type	Variable	Use	Description
char	**name-ptr*	update	pointer to the string to be verified.
int	*length*	input	length of string. Set this to −1 if the string pointed to by *name-ptr* is null-terminated.
int	*rc*	returned	return code: 0 = successful 1 = blank 2 = longer than eight characters 3 = first character invalid 4 = other invalid character.

Usage

```
rc = SAS_ZNAMVER(name-ptr, length);
```

Description

The SAS_ZNAMVER routine verifies that a name is a valid SAS name. To do this, the system first left-justifies and uppercases the string passed to it and then validates the transformed string. If any invalid characters are found, the return code indicates the type of problem found.

Example

The following example uses SAS_ZNAMVER to test a character string to determine if it is a valid SAS name. The example then uses the SAS_ZNAME routine to fix the string so it meets the requirements for a valid SAS name.

```
char  temp[8];

SAS_ZMOVEI("12345   ", temp, 8);
if (SAS_ZNAMVER(temp, 8)!=0)
    SAS_XPSLOG("%*s is NOT a valid SAS name", 8, temp);
SAS_ZNAME(temp);
if (SAS_ZNAMVER(temp, 8)==0)
    SAS_XPSLOG("%*s is a valid SAS name", 8, temp);
```

SAS_ZNDMOVI

Move memory nondestructively

Declarations

Type	Variable	Use	Description
ptr	*from-ptr*	input	pointer to the area to move from
ptr	*to-ptr*	output	pointer to the area to move to
int	*length*	input	the number of bytes to move

Usage

C	`SAS_ZNDMOVI(from-ptr, to-ptr, length);`
FORTRAN	`CALL SAS_ZNDMOVI(from-ptr, to-ptr, length)`
PL/I	`CALL SAS_ZNDMOVI(from-ptr, to-ptr, length);`

Description

The SAS_ZNDMOVI routine copies a specified number of bytes from one location to another. This routine should be used when the data areas pointed to by *to-ptr* and *from-ptr* may overlap. A nondestructive move preserves integrity in the source, while moving memory to the destination. SAS_ZNDMOVI copies the source string to a temporary storage area and then copies that string to the destination. Thus, if the two storage areas overlap, using SAS_ZNDMOVI prevents unwanted propagation of characters.

Example

The following example illustrates a nondestructive move:

```
char area[] = "abcdefghijklmn";
char *area_1, *area_2;

area_1 = area;
area_2 = area+3;
SAS_ZNDMOVI(area_1,area_2,5);
```

The contents of *area* after this move are as follows:

```
abcabcdeijklmn
```

SAS_ZPARM

Return a pointer to the *n*th element of a list of extra-long character strings (mode 5 of the @LIST semantic)

Declarations

Type	Variable	Use	Description
struct STLIST	**stmt-fld-ptr*	input	address of the statement field structure that contains the character string. Use the SFLD routine to obtain this pointer.
int	*number*	input	element number. Keep in mind the array is 0-based, so the third element is number 2.
int	**length-ptr*	output	length of that element.
char	**char-ptr*	returned	the pointer to the *n*th element in a character string.

Usage

```
char-ptr = SAS_ZPARM(stmt-fld-ptr, number, length-ptr);
```

Description

The SAS_ZPARM routine returns a pointer to a character string in the `clist` portion of the statement structure, which is created when your grammar uses the @LIST(*n*,5) semantic action. You can obtain this pointer by using the SFLD routine, described in Chapter 24, "Miscellaneous Routines and Values." This routine does not allocate any memory, it only returns a pointer to the location of the string.

If the *n*th element of the list does not exist, *length* is returned as 0, and *stmt-fld-ptr* is NULL.

Example

See the example for the SAS_ZPARMN routine.

SAS_ZPARMN

Return the number of elements in a list of extra-long character strings (mode 5 of the @LIST semantic)

Declarations

Type	Variable	Use	Description
struct STLIST	**stmt-fld-ptr*	input	address of statement field structure that contains the list of character strings. Use the SFLD routine to obtain this pointer.
int	*number*	returned	the number of elements in the list.

Usage

```
number = SAS_ZPARMN(stmt-fld-ptr);
```

Description

The SAS_ZPARMN routine returns the number of extra-long character strings stored in a `clist` portion of the statement structure, which is created when your grammar uses the @LIST(*n*,5) semantic action.

Example

The following example illustrates how to use the SAS_ZPARMN routine to access the number of extra-long character strings in a list. The example then uses the SAS_ZPARM routine to access the value of each character string.

```
struct STLIST  *p1;
int    i,
       n,
       len;
char   *p;


p1 = (struct STLIST *)SFLD(proc.head, 1);
n = SAS_ZPARMN(p1);
SAS_XPSLOG("This list has %d extra long character strings.", n);
for (i = 0; i < n; i++) {
   p = SAS_ZPARM(p1, i, &len);
   SAS_XPSLOG("  element %d is '%*s'.", i, len, p);
   }
```

SAS_ZPFMT

Determine number of decimal places for printing

Declarations

Type	Variable	Use	Description
double	*value*	input	number that needs to be formatted
int	*width*	input	width of field for printing number. This value must be greater than 0 and less than or equal to 32.
int	*decimals*	returned	number of decimal places to use in the format.

Usage

```
decimals = SAS_ZPFMT(value, width);
```

Description

The SAS_ZPFMT routine determines the number of decimal places to print to allow the maximum number of significant digits. One space is reserved for the sign even if the value is greater than 0.

Example

You can use SAS_ZPFMT to determine the decimal places to use to print a column of numbers so that the decimal points line up, as the following example illustrates:

```
   /* find the largest value in the array */
xmax = 0;
for (i = 1; i <= n, i++) {
   xmax = (xmax >= abs(x[i])) ? xmax : abs(x[i]);
}

   /* use SAS_ZPFMT to determine how many decimal places to print */
d = SAS_ZPFMT(xmax, 12);

print the array using the value determined by SAS_ZPFMT
```

SAS_ZPGT2

Compute the probability that the absolute value of a random variable from the Student's *t*-distribution is greater than or equal to *t*

Declarations

Type	Variable	Use	Description
double	*t2*	input	the square of the random variable *t*, with $t*t >= 0$
long	*df*	input	the degrees of freedom, with $df>0$
double	*y*	returned	the computed probability that the absolute value of a random variable from the Student's *t*-distribution is greater than or equal to *t*. Note that $0<=y<=1$. If *y* is -2, the algorithm failed.

Usage

```
y = SAS_ZPGT2(t2, df);
```

Description

The SAS_ZPGT2 routine computes the probability that the absolute value of a random variable from the Student's *t*-distribution is greater than or equal to *t*.

The SAS_ZPGT2 routine is related to the incomplete beta distribution by the following:

$$\text{SAS\_ZPGT2}(t2,df) = I_{\frac{df}{df+t2}}\left(\frac{df}{2},\frac{1}{2}\right)$$

where for x, a, and b, the probability from the incomplete beta distribution is given by

$$I_x(a,b) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} \int_0^x u^{a-1}(1-u)^{b-1}du \quad .$$

SAS_ZPGX2

Compute the probability that the value of a random variable from the chi-square distribution is greater than or equal to x2

Declarations

Type	Variable	Use	Description
double	*x2*	input	the value of the random variable, with *x2*>= 0.
long	*df*	input	the degrees of freedom, with *df*> 0.
double	*y*	returned	the computed probability that the random variable from the chi-square distribution is greater than or equal to *x2*. Note that 0<=y<=1. If y is equal to MACMISSING, the algorithm failed.

Usage

```
y = SAS_ZPGX2(x2, df);
```

Description

The SAS_ZPGX2 routine computes the probability that the value of a random variable from the chi-square distribution is greater than or equal to *x2*. The value returned by SAS_ZPGX2 is given by the following:

$$\text{SAS\_ZPGX2}(x2,df) = 1 - \Gamma\left(\frac{x2}{2}, \frac{df}{2}\right);$$

where

$$\Gamma(x,a) = \frac{1}{\Gamma(a)} \int_0^x u^{a-1} e^{-u} du$$

is the probability that the random variable from the gamma distribution with shape parameter *a* is less than or equal to *x*.

SAS_ZPRBF

Compute the probability that the value of a random variable from the *F*-distribution is greater than or equal to *f*

Declarations

Type	Variable	Use	Description
double	*f*	input	the value of the random variable, with $f>=0$.
double	*ndf*	input	the numerator degrees of freedom, with $ndf>=0$.
double	*ddf*	input	the denominator degrees of freedom, with $ddf>=0$.
double	*y*	returned	the computed probability that the value of a random variable from the *F*-distribution is greater than or equal to *f*. Note that $0<=y<=1$. If y is -2, the algorithm failed.

Usage

```
y = SAS_ZPRBF(f, ndf, ddf);
```

Description

The SAS_ZPRBF routine computes the probability that the value of a random variable from the *F*-distribution is greater than or equal to *f*. The SAS_ZPRBF routine is related to the incomplete beta function by the following:

$$\text{SAS\_ZPRBF}(f,ndf,ddf) = I_{\frac{ddf}{ddf+f*ndf}}\left(\frac{ddf}{2}, \frac{ndf}{2}\right) .$$

SAS_ZPRBFMT

Format a probability value

Declarations

Type	Variable	Use	Description
double	*y*	input	the probability value. Use a negative value to print an exact 0 (instead of 0.0001).
ptr	*value-ptr*	output	a pointer to a six-character formatted value.

Usage

C	`SAS_ZPRBFMT(y, value-ptr);`
FORTRAN	`CALL SAS_ZPRBFMT(y, value-ptr)`
PL/I	`CALL SAS_ZPRBFMT(y, value-ptr);`

Description

The SAS_ZPRBFMT routine formats a probability value for printing. This routine uses the value of the PROBSIG= system option to determine the number of significant digits to print.

SAS_ZPRBNRM

Compute the probability from the normal distribution

Declarations

Type	Variable	Use	Description
double	*x*	input	must be positive.
double	*y*	returned	the computed probability at *x*. Note that $0<=y<=1$.

Usage

```
y = SAS_ZPRBNRM(x);
```

Description

The SAS_ZPRBNRM routine computes the probability from the normal distribution.

SAS_ZPROBIT

Compute the quantile from the normal distribution

Declarations

Type	Variable	Use	Description
double	*x*	input	the probability, with 1.e−16<x<1.−MACEPS
double	*y*	returned	the computed quantile. Note that, in theory $-\infty<y<\infty$, but the actual implementation returns −8.222082216130435612 if $x<$1.e−16 and 7.941345326170996780 if $x>1-$MACEPS.

Usage

```
y = SAS_ZPROBIT(x);
```

Description

The SAS_ZPROBIT routine computes the quantile from the normal distribution.

SAS_ZPSIG

Determine the format for printing a number with a given number of significant digits

Declarations

Type	Variable	Use	Description
double	*number*	input	the number to determine the format for. This may be a missing value.
int	*significant*	input	the number of significant digits desired.
int	*max-width*	input	the maximum field width. Must be less than or equal to 32.
int	**width-ptr*	output	pointer to the field width.
int	**decimals-ptr*	output	pointer to the number of decimals: 99 = *max-width* is too small to print *significant* number of digits. In this case, *width* is set to *max-width*.

Usage

C	`SAS_ZPSIG(number, significant, max-width, width-ptr, decimals-ptr);`
FORTRAN	`CALL SAS_ZPSIG(number, significant, max-width, width-ptr, decimals-ptr)`
PL/I	`CALL SAS_ZPSIG(number, significant, max-width, width-ptr, decimals-ptr);`

Description

The SAS_ZPSIG routine determines the format that should be used to print a number with a specified number of significant digits. You may want to use the SAS_XOPTIGT routine to access the value of the PROBSIG= SAS system option, and then use that value for the *significant* parameter.

SAS_ZPSIG2

Determine the format for printing two numbers so that each has a given number of significant digits

Declarations

Type	Variable	Use	Description
double	*num-1*	input	the first number to determine format for. This may be a missing value.
double	*num-2*	input	the second number to determine format for. This may be a missing value.
int	*significant*	input	the number of significant digits desired.
int	*max-width*	input	the maximum field width. Must be less than or equal to 32.
int	**width-ptr*	output	pointer to the field width.
int	**decimals-ptr*	output	pointer to the number of decimals.
double	**small-ptr*	output	pointer to the smallest number that can be printed with *width* and *decimals* and still give *significant* number of digits: 99 = *max-width* is too small to print *significant* number of digits for both of the numbers. In this case, *width* is set to *max-width* and *small* is set to 0.

SAS_ZPSIG2 *continued*

Usage

C `SAS_ZPSIG2(num-1, num-2, significant, max-width, width-ptr, decimals-ptr, small-ptr);`

FORTRAN `CALL SAS_ZPSIG2(num-1, num-2, significant, max-width, width-ptr, decimals-ptr, small-ptr)`

PL/I `CALL SAS_ZPSIG2(num-1, num-2, significant, max-width, width-ptr, decimals-ptr, small-ptr);`

Description

The SAS_ZPSIG2 routine determines the format for printing two numbers so that both numbers have a specified number of significant digits. One space is always allowed for the sign. If only one of the numbers can be printed with the *signficant* number of digits, then *width* and *decimals* are computed for the larger number. Usually *num-1* and *num-2* are the maximum and minimum absolute values in a vector to be printed.

Example

The following example illustrates printing one element of the vector:

```
double  xmax, xmin, small,
        x(|n|) = {values } ;
int     sig, maxw, w, d, i;
long    fcode;

xmax = 0;
xmin = 0;
sig = 10;
maxw = 32;
for (i = 0; i<n; i++) {

      /* get the maximum and minimum values */
   xmax = (xmax >= abs(x(|i|))) ? xmax : abs(x(|i|));
   xmin = (xmin <= abs(x(|i|))) ? xmin : abs(x(|i|));
}

   /* obtain format code for scientific notation */
SAS_XFNAME("E", 'F', NULL, &fcode);

   /* determine width and decimal places */
SAS_ZPSIG2(xmax, xmin, sig, maxw, &w, &d, &small);

   /* print using w.d or E notation */
for (i = 0; i<n; i++) {
   if (abs(x(|i|)) < small) {
      SAS_XPS("%*.*f", w, d, x(|i|));
   }
```

```
    else {
       SAS_XPS("%*.$f", w, fcode, x(|i|));
    }
    SAS_XPSKIP(1);
}
```

SAS_ZQC

Compute the quantile from the chi-square distribution

Declarations

Type	Variable	Use	Description
double	*p*	input	the lower probability, with $0<p<1$
double	*df*	input	the degrees of freedom, with $df>0$
double	**q*	output	the returned quantile
int	*rc*	returned	the return code: 0 = successful 1 = unsuccessful

Usage

```
rc = SAS_ZQC(p, df, q);
```

Description

The SAS_ZQC routine computes the quantile from the chi-square distribution. The quantile q from the chi-square distribution, with degrees of freedom ν, is the solution of the following nonlinear equation:

$$\Gamma\left(\frac{q}{2}, 0.5\nu\right) = p$$

where $\Gamma(.,.)$ is the probability from the incomplete gamma distribution. The solution is computed iteratively from a rough initial estimate. This estimate is then refined by using Newton's method. With extreme cases, the convergence of the Newton's iterates cannot be always guaranteed.

SAS_ZQF

Compute the quantile from the *F*-distribution

Declarations

Type	Variable	Use	Description
double	*p*	input	the lower probability, with $0<p<1$
double	*ndf*	input	the numerator degrees of freedom, with $ndf>0$
double	*ddf*	input	the denominator degrees of freedom, with $ddf>0$
double	**q*	output	the returned quantile
int	*rc*	returned	the return code: 0 = successful 1 = unsuccessful

Usage

```
rc = SAS_ZQF(p, ndf, ddf, q);
```

Description

The SAS_ZQF routine computes the quantile from the *F*-distribution. The quantile q from the *F*-distribution, with ν_n numerator degrees of freedom and ν_d denominator degrees of freedom, is the solution of the following nonlinear equation:

$$I_{\frac{\nu_n q}{\nu_d+\nu_n q}}(0.5\nu_n, 0.5\nu_d) = p$$

where $I_.(.,.)$ is the probability from the incomplete beta distribution. The solution is computed iteratively from a rough initial estimate. This estimate is then refined by using Newton's method. With extreme cases, the convergence of the Newton's iterates cannot be always guaranteed.

SAS_ZQT

Compute the quantile from the Student's *t*-distribution

Declarations

Type	Variable	Use	Description
double	*p*	input	the lower probability, with $0<p<1$
double	*df*	input	the degrees of freedom, with $df>0$
double	**q*	output	the returned quantile
int	*rc*	returned	the return code: 0 = successful 1 = unsuccessful

Usage

```
rc = SAS_ZQT(p, df, q);
```

Description

The SAS_ZQT routine computes the quantile from the Student's *t*-distribution. The quantile *q* from the t-distribution, with degrees of freedom ν, is the solution of the following nonlinear equation:

$$1 + \text{sign}(q)I_{\frac{q^2}{\nu + q^2}}(0.5, 0.5\nu) = 2p$$

where $I_.(.,.)$ is the probability from the incomplete beta distribution. The solution is computed iteratively from a rough initial estimate. This estimate is then refined by using Newton's method. With extreme cases, the convergence of the Newton's iterates cannot be always guaranteed.

SAS_ZRANUNI

Generate a random variate from a uniform distribution

Declarations

Type	Variable	Use	Description
long	**seed-ptr*	update	a pointer to the seed required to start and maintain the stream of random numbers
double	*number*	returned	the generated number

SAS_ZRANUNI *continued*

Usage

```
number = SAS_ZRANUNI(seed-ptr);
```

Description

The SAS_ZRANUNI function returns a number generated from the uniform distribution on the interval (0,1) using a prime modulus multiplicative generator with modulus $2^{31}-1$ and multiplier 397204094 (Fishman and Moore 1982). The SAS_ZRANUNI routine also updates the value pointed to by *seed-ptr* to prepare for a subsequent call.

SAS_ZRSDBL

Restore a float value from a value that can be sorted

Declarations

Type	Variable	Use	Description
ptr	*value-ptr*	update	pointer to the value that can be sorted. SAS_ZRSDBL changes the field pointed to by this pointer to contain the restored float value.
int	*length*	input	length in bytes of the float value (2-8).
long	*rc*	returned	return code: 0 = successful.

Usage

```
rc = SAS_ZRSDBL(value-ptr, length);
```

Description

The SAS_ZRSDBL routine restores a floating-point number from a coded form of the number that can be sorted as if it were a character variable. The conversion is performed in place; that is, the value pointed to by *value-ptr* is changed. To create coded values, use the SAS_ZCSDBL routine.

Example

The following example converts characters that can be sorted into double-precision floating-point values:

```
char    temp[200];
double  value;

value = 1;
SAS_ZMOVEI(&value, temp, sizeof(double));
SAS_ZCSDBL(temp, sizeof(double));
SAS_ZRSDBL(temp,sizeof(double));
SAS_ZMOVEI(temp,&value,sizeof(double));
SAS_XPSLOG("Restored collatable double should be 1.00: %f.",value);
```

SAS_ZRSINT

Restore an integer from a value that can be sorted

Declarations

Type	Variable	Use	Description
int, long, or short	*value*	update	value that can be sorted. After calling SAS_ZRSINT, this field contains the restored integer value.
int	*length*	input	length in bytes of the integer value. Valid values are: sizeof (int) sizeof (long) sizeof (short)
long	*rc*	returned	return code: 0 = successful.

Usage

C	`rc = SAS_ZRSINT((ptr) &int_value, length);`
FORTRAN	`rc = SAS_ZRSINT(IADDR(int_value), length)`
PL/I	`rc = SAS_ZRSINT(ADDR(int_value), length);`

Description

The SAS_ZRSINT routine restores an integer from a coded form of the number that can be sorted as if it were a character variable. The conversion is performed in place; that is, the value pointed to by *value-ptr* is changed. To create coded values, use the SAS_ZCSINT routine.

SAS_ZRSINT *continued*

Example

The following example converts an integer into a character value and then restores the original integer value:

```
int    ivalue;
ptr    temp;

ivalue = 1;
SAS_ZMOVEI(&ivalue,temp,sizeof(int));
SAS_ZCSINT(temp,sizeof(int));
SAS_ZRSINT(temp,sizeof(int));
SAS_ZMOVEI(temp,&ivalue,sizeof(int));
SAS_XPSLOG("Restored collatable integer should be 1: %d.",ivalue);
```

SAS_ZRTCHR

Restore a transport character string to local format

Declarations

Type	Variable	Use	Description
ptr	*source-ptr*	input	address of transport data to restore
int	*length*	input	length of data to restore
char	**dest-ptr*	output	address of storage area for local character string
long	*rc*	returned	return code: 0=successful

Usage

```
rc = SAS_ZRTCHR(source-ptr, length, dest-ptr);
```

Description

The SAS_ZRTCHR routine converts a character string in transport format to a string on the local host. You cannot change a value in transport format. You can only store it or move it. Therefore, you need to convert a value before you manipulate it.

Example

See the example for the SAS_ZCTCHR routine.

SAS_ZRTDBL

Restore a transport double to local format

Declarations

Type	Variable	Use	Description
ptr	*source-ptr*	input	pointer to the transport double to restore to the local format
int	*length*	input	length of transport double (2–8 bytes)
double	*value*	returned	value restored to double in the local host environment

Usage

```
value = SAS_ZRTDBL(source-ptr, length);
```

Description

The SAS_ZRTDBL routine converts a double in transport format to the appropriate value on the local host. You cannot change a value in transport format. You can only store it or move it. Therefore, you need to convert a value before you manipulate it.

If the value pointed to by *source-ptr* is too large, *value* is set to MACBIG for positive numbers or −MACBIG for negative numbers. If the value pointed to by *source-ptr* is too small to be represented accurately, *value* is set to MACSMALL for positive numbers or −MACSMALL for negative numbers.

Example

See the example for the SAS_ZCTDBL routine.

SAS_ZRTINT

Restore a transport format integer to local format

Declarations

Type	Variable	Use	Description
ptr	*source-ptr*	input	address of transport format integer to restore to local format
int	*length*	input	length of transport integer
long	*value*	returned	value restored to integer in the local host environment

Usage

```
value = SAS_ZRTINT(source-ptr, length);
```

Description

The SAS_ZRTINT routine converts an integer in transport format to the appropriate value on the local host. You cannot change a value in transport format. You can only store it or move it. Therefore, you need to convert a value before you manipulate it.

Example

See the example for the SAS_ZCTINT routine.

SAS_ZSCP

Accumulate the cross product of multiples of rank 1 matrices

Declarations

Type	Variable	Use	Description
double	**v*	input	a pointer to the one dimensional array, **v**
int	*nv*	input	the size of the vector, **v**.
double	*w*	input	the scale
double	**m*	output	a pointer to the square symmetric matrix, **M**, stored in compressed mode
int	*l*	returned	the returned value: 0 = one of the components of **v** is missing or *w* is missing or 0 1 = successful

Usage

```
l = SAS_ZSCP(v, nv, w, m);
```

Description

The SAS_ZSCP routine accumulates the cross product of multiples of rank 1 matrices. Let $\mathbf{v}_i$ be $k+1$ vectors of size *nv*. Consider computing the sum of multiples of the cross (outer) products of the following form:

$$\mathbf{M}_k = \sum_{i=0}^{i=k} w_i \mathbf{v}_i \mathbf{v}_i' \quad .$$

Note that **M** is a symmetric matrix, so it is efficient to store **M** in compressed mode. Partial sums can be accumulated by starting with $\mathbf{M}_0=0$ (in compressed mode) and using the following equation:

$$\mathbf{M}_l = \mathbf{M}_{l-1} + w_l \mathbf{v}_l \mathbf{v}_l'$$

where $l = 1, \ldots, k$. Multiple calls to SAS_ZSCP accumlate the cross (outer) product in the matrix **M**.

SAS_ZSTOL

Scan character string for an integer and convert

Declarations

char	**string-ptr*	input	address of the character string.
int	*length*	input	length of the character string. Specify −1 if the string is null-terminated.
int	*signed*	input	flag indicating whether integer must be unsigned or not: 0 = require unsigned 1 = allow sign.
int	**number-ptr*	output	pointer to the number of characters read.
long	**value-ptr*	output	long integer to be derived from the string.
int	*rc*	returned	return code: 0 = successful otherwise = unsuccessful.

Usage

```
rc =  SAS_ZSTOL(string-ptr, length, signed, number-ptr,
                value-ptr);
```

Description

The SAS_ZSTOL routine converts a character string to a long integer. If the string does not represent a number, a nonzero return code is returned. If the string is numeric but outside the range of a long-integer, a nonzero return code is returned.

Leading and trailing blanks are permitted. All values to the right of a decimal point are ignored.

The following table illustrates some values returned by this routine:

Value pointed to by *string-ptr*	Value pointed to by *value-ptr*	Value pointed to by *number-ptr*
1.2	1	1
1E2	1	1
123X	123	3

Example

The following example converts a character string to a long integer:

```
int   nchar,
      i;
long  value;

i = SAS_ZSTOL("123XY", 5, 0, &nchar, &value);
SAS_XPSLOG("SAS_ZSTOL returned nchar=%d value=%ld rc=%d. (Should be
nchar=3 value=123 rc=0).", nchar, value, i);
```

SAS_ZSTRANC

Substitute one character for another

Declarations

Type	Variable	Use	Description
char	**string-ptr*	update	string to be processed (may be NULL)
int	*length*	input	length of string; −1 if null terminated
int	*to*	input	the new character
int	*from*	input	the character to be changed

Usage

C	`SAS_ZSTRANC(string-ptr, length, to, from);`
FORTRAN	`CALL SAS_ZSTRANC(string-ptr, length, to, from)`
PL/I	`CALL SAS_ZSTRANC(string-ptr, length, to, from);`

Description

The SAS_ZTRANC routine substitutes one character for another in a character string. The substitution is made in place; that is, the value pointed to by *string-ptr* is changed.

SAS_ZSTRANC *continued*

Example

The following example translates all occurrences of the character X to the character −.

```
char  temp[5];

SAS_ZMOVEI("AXBXC", temp, 5);
SAS_ZSTRANC(temp, 5, '-', 'X');
SAS_XPSLOG("SAS_ZSTRANC modified AXBXC to be %5s (should be A-B-C).",
           temp);
```

SAS_ZSTRANS

Convert characters in a string

Declarations

Type	Variable	Use	Description
char	**string-ptr*	update	pointer to the string to be processed (may be NULL).
int	*length*	input	length of *string*; −1 if null terminated.
char	**to-ptr*	input	pointer to the new characters. The characters in this string should match one-to-one for the characters in the *from* string.
char	**from-ptr*	input	pointer to characters to be changed. The characters in this string should match one-to-one for the characters in the *to* string.
int	*to-from-length*	input	length of *from* and *to* strings, >=0.

Usage

C `SAS_ZSTRANS(string-ptr, length, to-ptr, from-ptr, to-from-length);`

FORTRAN `CALL SAS_ZSTRANS(string-ptr, length, to-ptr, from-ptr, to-from-length)`

PL/I `CALL SAS_ZSTRANS(string-ptr, length, to-ptr, from-ptr, to-from-length);`

Description

The SAS_ZTRANS routine substitutes the characters in one string for the characters in another string. The two strings should have a matching number of characters. The substitution occurs for corresponding characters in the two strings. The substitution is made in place; that is, the value pointed to by *string-ptr* is changed.

Example

The following example translates all occurrences of the character X to − and all occurrences of the character C to /.

```
char  temp[5];

SAS_ZMOVEI("AXBXC", temp, 5);
SAS_ZSTRANS(temp, 5, "-/", "XC", 2);
SAS_XPSLOG("SAS_ZSTRANS modified AXBXC to be %5s (should be A-B-/).",
           temp);
```

SAS_ZSTRCOM

Compare two strings with blank padding on shorter

Declarations

Type	Variable	Use	Description
char	**string-1-ptr*	input	string to compare (may be NULL).
int	*length-1*	input	length of the first string. Specify −1 if the string is null terminated.
char	**string-2-ptr*	input	string to compare (may be NULL).
int	*length-2*	input	length of the second string. Specify −1 if the string is null terminated.
int	*compare*	returned	number indicating how the strings compared. Values are 0=indicates string 1 and string 2 are equal. <0=indicates that string 1 is less than string 2. That is, string 1 would appear first in a sorted sequence. The absolute value of *compare* is the position of the first different character in the two strings. >0=indicates that string 1 is greater than string 2. That is, string 1 would appear second in a sorted sequence. The value of *compare* is the position of the first different character in the two strings.

Usage

```
compare = SAS_ZSTRCOM(string-1-ptr, length-1, string-2-ptr, length-2);
```

Description

The SAS_ZSTRCOM routine compares two character strings and indicates which of the two strings appears first in a sorted sequence. If the strings are not of equal length, the shorter string is treated as if it is padded with blanks to the length of the longer string.

Example

The following example compares two character strings of different lengths by padding the shorter string with blanks:

```
int   i;

i = SAS_ZSTRCOM("ABC  ", 5, "ABC", 3);
SAS_XPSLOG("SAS_ZSTRCOM compared like strings: %d (should be 0).", i);
```

SAS_ZSTRDEL

Delete characters from a string

Declarations

Type	Variable	Use	Description
char	**string-ptr*	update	pointer to string to be processed (may be NULL).
int	*length*	input	length of input string. Specify -1 if the string is null-terminated.
char	**delete-ptr*	input	pointer to characters to be deleted (may be NULL).
int	*delete-len*	input	length of string of characters to delete. Specify -1 if the string is null terminated.
int	*new-len*	returned	length of string after deleting characters. Trailing blanks and null terminators are not included in the length.

Usage

```
new-len = SAS_ZSTRDEL(string-ptr, length, delete-ptr, delete-len);
```

Description

The SAS_ZSTRDEL routine deletes characters included in one string from another string. Characters in the string pointed to by *string-ptr* that occur in the string pointed to by *delete-ptr* are deleted.

Example

The following example compresses all the characters A out of the input string.

```
char  temp[8];
int   i;

SAS_ZMOVEI("XXAXXXAXX", temp, 8);
i = SAS_ZSTRDEL(temp, 8, "A", 1);
SAS_XPSLOG("SAS_ZSTRDEL created compressed string '%*s'
(should be 'XXXXXX').", i, temp);
```

SAS_ZSTRIP

Return string length without the trailing blanks

Declarations

Type	Variable	Use	Description
char	**string-ptr*	input	pointer to string to process (may be NULL).
int	*length*	input	length of string. Specify −1 if the string is null terminated.
int	*no-blanks*	returned	length of string without trailing blanks.

Usage

```
no-blanks = SAS_ZSTRIP(string-ptr, length);
```

Description

The SAS_ZSTRIP routine returns the length of a string without including blank padding at the end of the string. Embedded blanks are counted in the length.

Example

The following example returns of the length of a character string without including the trailing blanks:

```
int   i;

i = SAS_ZSTRIP("ABC     ", -1);
SAS_XPSLOG("SAS_ZSTRIP says 'ABC     ' has nonblank length of %d.
(should be 3).", i);
```

SAS_ZSTRIPQ

Remove quotes from a null-terminated string

Declarations

Type	Variable	Use	Description
char	**in-string-ptr*	input	pointer to the input string. This string must be null terminated.
char	**out-string-ptr*	input	pointer to the output string. This string is null terminated.

Usage

C	`SAS_ZSTRIPQ(in-string-ptr, out-string-ptr);`
FORTRAN	`CALL SAS_ZSTRIPQ(in-string-ptr, out-string-ptr)`
PL/I	`CALL SAS_ZSTRIPQ(in-string-ptr, out-string-ptr);`

Description

This routine removes quotation marks from the beginning, ending, and middle of a character string. If the string has two quotation marks in a row within the string, only one of the two marks is removed.

Note: This routine can actually remove any character from a string. The first character of the string is the delimiter that is stripped from the rest of the string.

The output string has a maximum length of 256 characters not including the null terminator. If the output string contains no characters, only a null terminator is placed in the target string.

Example

The following example removes the single quotation marks from a character string:

```
char  temp[20];

SAS_ZSTRIPQ("'IT''S A STRING'", temp);
SAS_XPSLOG("SAS_ZSTRIPQ created %s (should be IT'S A STRING).", temp);
```

SAS_ZSTRJLS

Perform justification on strings

Declarations

Type	Variable	Use	Description
char	**string-ptr*	update	pointer to the string to justify (NULL treated as blanks).
int	*length*	input	length of string. Specify −1 if the string is null terminated.
int	*type*	input	type of justification. For the PL/I and FORTRAN languages, declare this variable as a 1-byte character. Valid values are 'l' = left justification 'r' = right justification 'c' = centering 'n' = no justification.
int	*to-length*	output	returned length of string. This length includes the first nonblank character through the last nonblank character. The address of this integer can be replaced by NULL if you do not want to obtain this information.
int	*to-start*	output	the position of the first nonblank character in *string* after justification. This number is a 0-based index for the character in the string. The address of this integer can be replaced by NULL if you do not want to obtain this information.

Usage

C	`SAS_ZSTRJLS(string-ptr, length, type, &to-length, &to-start);`
FORTRAN	`CALL SAS_ZSTRJLS(string-ptr, length, type, IADDR(to-length), IADDR(to-start))`
PL/I	`CALL SAS_ZSTRJLS(string-ptr, length, type, ADDR(to-length), ADDR(to-start));`

Description

The SAS_ZSTRJLS routine right-or left-justifies or centers a string in the specified string length. When the string is completely blank, the *length* and *to-start* fields are set to 0.

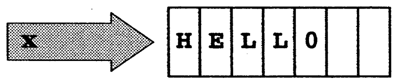
Example

The following example illustrates the effects of this routine:

```
SAS_ZMOVEI("  HELLO", x, 7);
SAS_ZSTRJLS(x, 7, 'l', &l, &s);
```

The result of these statements is that $l=5$, $s=0$, and x points to the string illustrated in Figure 23.1.

Figure 23.1
Left-Justifying a String

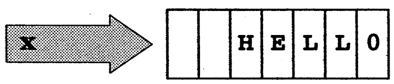


In this example, x points to a left-justified string that is right-justified by SAS_ZSTRJLS.

```
SAS_ZMOVEI("HELLO  ", x, 7);
SAS_ZSTRJLS(x, 7, 'r', NULL, NULL);
```

The result of these statements is illustrated in Figure 23.2.

Figure 23.2
Right-Justifying a String



Example

The following example left-justifies a string:

```
char  temp[13];
int   i;

SAS_ZMOVEI("   TO LEFT   ", temp, 13);
SAS_ZSTRJLS(temp, 13, 'l', &i, NULL);
SAS_XPSLOG("SAS_ZSTRJLS left-justified: '%13s' l=%d.
(should be 'TO LEFT      ' l=7).", temp, i);
```

SAS_ZSTRLEN

Return length of null-terminated string

Declarations

Type	Variable	Use	Description
char	**string-ptr*	input	pointer to the string to search for null terminator
int	*length*	returned	length of string

Usage

```
length = SAS_ZSTRLEN(string-ptr);
```

Description

The SAS_ZSTRLEN routine returns the offset to the null terminator for a null terminated string. The string must be null terminated. If not, the routine continues searching memory until it finds a null terminator and then returns the offset to that location.

Example

The following example returns the length of the input string and prints the length using SAS_XPSLOG:

```
SAS_XPSLOG("SAS_ZSTRLEN(
ABCDE
) returns %d. (should be 5).",
          SAS_ZSTRLEN("ABCDE"));
```

SAS_ZSTRLO

Convert a string to lowercase (in place)

Declarations

Type	Variable	Use	Description
char	**string-ptr*	update	pointer to string to process (may be NULL).
int	*length*	input	length of string to convert. If you want to convert only part of the string, specify only the length of the characters to convert. Specify −1 if the string is null terminated and you want to convert the entire string.

Usage

C	`SAS_ZSTRLO(string-ptr, length);`
FORTRAN	`CALL SAS_ZSTRLO(string-ptr, length)`
PL/I	`CALL SAS_ZSTRLO(string-ptr, length);`

Description

The SAS_ZSTRLO routine converts uppercase characters to lowercase characters in a string. The substitution is made in place; that is, the value pointed to by *string-ptr* is changed.

Example

The following example converts all the letters in the input string to lowercase letters:

```
char  temp[23];

SAS_ZMOVEI("Mixed Case String 12345", temp, 23);
SAS_ZSTRLO(temp, 23);
SAS_XPSLOG("SAS_ZSTRLO generates '%23s'
(should be 'mixed case string 12345').", temp);
```

SAS_ZSTRMOV

Move a string with blank-padding or truncation

Declarations

Type	Variable	Use	Description
char	**from-ptr*	input	pointer to string to be copied (may be NULL).
int	*from-length*	input	length of string. Specify −1 if the string is null terminated and you want to move the entire string.
char	**to-ptr*	output	where to copy the string to.
int	*to-length*	input	length of area to receive string.
int	*length*	returned	length of the string pointed to by *to-ptr*, not including blank padding.

Usage

```
length = SAS_ZSTRMOV(from-ptr, from-length, to-ptr, to-length);
```

Description

The SAS_ZSTRMOV routine copies one area of memory to another and truncates or blank pads as necessary. If the area pointed to by *to-ptr* overlaps the area pointed to by *from-ptr*, the move is performed nondestructively; that is, unwanted propagation of values does not occur.

The length of the string that is moved is *to-length* unless *from-length* is shorter. When *from-length* is shorter, the value pointed to by *to-ptr* is either blank padded to *to-length* or null terminated, depending on whether *to-length* is a positive or negative value.

Example

The following examples illustrate using SAS_ZSTRMOV:

- If `from_ptr` points to A followed by a null terminator, the following call changes the area pointed to by `to_ptr` to contain A followed by 7 blanks:

```
SAS_ZSTRMOV(from_ptr, -1, to_ptr, 8);
```

- If you change the last argument to a negative value, as in the following example, `to_ptr` points to an area containing A, followed by a null terminator and 6 unused bytes.

```
SAS_ZSTRMOV(from_ptr, -1, to_ptr, -8);
```

- When `from_ptr` points to "ABCDEF", the following call changes the value pointed to by `to_ptr` to "ABCDEF ":

```
SAS_ZSTRMOV(from_ptr, 6, to_ptr, 8);
```

- To truncate a string and add a null terminator, use SAS_ZSTRMOV as follows:

```
SAS_ZSTRMOV (from_ptr, 6, to_ptr, -4);
```

  As a result of this call, `to_ptr` points to "ABC" followed by a null terminator.

SAS_ZSTRNDX

Search for one string inside another

Declarations

Type	Variable	Use	Description
char	**string-ptr*	input	pointer to the string to be searched (may be NULL).
int	*length*	input	length of string to search. Specify -1 if the string is null terminated.
char	**find-ptr*	input	pointer to the string to be searched for (may be NULL).
int	*find-length*	input	length of string to search for. Specify -1 if the string is null terminated.
int	*position*	returned	the 0-based offset of the first occurrence in the string pointed to by *string-ptr* of the *find-ptr* string: -1 = the second string is not found in the first.

Usage

```
position = SAS_ZSTRNDX(string-ptr, length, find-ptr, find-length);
```

Description

The SAS_ZSTRNDX routine searches the string pointed to by *string-ptr* to find the string pointed to by *find-ptr*. If *find-length* is 0, the *position* is set to 0. This routine is similar to SAS_ZSTRPOS, but SAS_ZSTRNDX permits you to specify more than one character.

SAS_ZSTRNDX *continued*

Example

The following example searches for the occurrence of the characters XYZ in the input string. Note that the value returned is a 0-based offset, so the fifth character in the string, X, is at location 4.

```
int   i;

i = SAS_ZSTRNDX("ABC XYZ QQQ", -1, "XYZ", 3);
SAS_XPSLOG("SAS_ZSTRNDX returns %d for index (should be 4).", i);
```

SAS_ZSTRNOT

Search for a character not equal to a given character

Declarations

Type	Variable	Use	Description
char	**string-ptr*	input	pointer to the string to search (may be NULL).
int	*length*	input	length of string to search for. Specify −1 if the string is null terminated.
int	*char*	input	character used in search.
int	*position*	returned	the 0-based offset of the first occurrence in the string pointed to by *string-ptr* of a character that is not *char*: −1 = *char* is not found in the string.

Usage

```
position = SAS_ZSTRNOT(string-ptr, length, char);
```

Description

The SAS_ZSTRNOT routine searches the string pointed to by *string-ptr* for characters other than the value of *char*. Using this routine is more efficient than using SAS_ZSTRVER when you are specifying a single character.

Example

The following example searches for the first occurrence of a character in the input string that is not X. Note that the value returned is a 0-based offset, so the sixth character in the string, Q, is at location 5.

```
int   i;

i = SAS_ZSTRNOT("XXXXXQXXX", -1, 'X');
SAS_XPSLOG("SAS_ZSTRNOT verifies %d as position (should be 5).", i);
```

SAS_ZSTRPOS

Search for a character in a string

Declarations

Type	Variable	Use	Description
char	**string-ptr*	input	pointer to the string to search (may be NULL).
int	*length*	input	length of string to search for. Specify −1 if the string is null-terminated.
int	*char*	input	character used in search.
int	*position*	returned	the 0-based offset of the first occurrence of *char* in the string pointed to by *string-ptr*: −1 = *char* is not found in the string.

Usage

```
position = SAS_ZSTRPOS(string-ptr, length, char);
```

Description

The SAS_ZSTRPOS routine searches the string pointed to by *string-ptr* to find the first occurrence of *char*. Using this routine is more efficient than using SAS_ZSTRNDX when you are searching for a single character in a string.

Example

The following example searches for the occurrence of the character A in the input string. Note that the value returned is a 0-based offset, so the first character in the string, A, is at location 0.

```
int   i;

i = SAS_ZSTRPOS("A B C D E", -1, 'A');
SAS_XPSLOG("SAS_ZSTRPOS returns %d for index (should be 0).", i);
```

SAS_ZSTRUP

Convert a string to uppercase in place

Declarations

Type	Variable	Use	Description
char	**string-ptr*	update	pointer to the string to convert to uppercase (may be NULL).
int	*length*	input	length of string to convert. If you want to convert only part of the string, specify only the length of the characters to convert. Specify −1 if the string is null terminated and you want to convert the entire string.

Usage

C	`SAS_ZSTRUP(string-ptr, length);`
FORTRAN	`CALL SAS_ZSTRUP(string-ptr, length)`
PL/I	`CALL SAS_ZSTRUP(string-ptr, length);`

Description

The SAS_ZSTRUP routine converts lowercase characters to uppercase characters in a string. The substitution is made in place; that is, the value pointed to by *string-ptr* is changed.

Example

The following example converts all the letters in the input string to uppercase letters.

```
char  temp[23];

SAS_ZMOVEI("Mixed Case String 12345", temp, 23);
SAS_ZSTRUP(temp, 23);
SAS_XPSLOG("SAS_ZSTRUP generates '%23s'
(should be 'MIXED CASE STRING 12345').", temp);
```

SAS_ZSTRVER

Search a string for any characters not in a second string

Declarations

Type	Variable	Use	Description
char	**string-ptr*	input	pointer to the string to search (may be NULL).
int	*length*	input	length of string to search. Specify −1 if the string is null terminated.
char	**exclude-ptr*	input	pointer to the string of characters to exclude from the search.
int	*exclude-length*	input	length of string of characters to exclude. Specify −1 if the string is null terminated.
int	*position*	returned	the 0-based offset of the first occurrence in the string pointed to by *string-ptr* of a character that is not in the string pointed to by *exclude-ptr*: −1 = the string pointed to by *string-ptr* contains only the characters in the second string.

Usage

```
position = SAS_ZSTRVER(string-ptr, length, exclude-ptr, exclude-length);
```

Description

The SAS_ZSTRVER routine searches the string pointed to by *string-ptr* for characters that *do not* occur in the string pointed to by *exclude-ptr*. Do not use this routine if *exclude-length* is 0 or *exclude-ptr* is NULL. This routine is similar to SAS_ZSTRNOT, but SAS_ZSTRVER permits you to specify more than one character.

Example

The following example searches for the first occurrence of a character in the input string that is not in the string ABCD. Note that the value returned is a 0-based offset, so the ninth character in the string, E, is at location 8. In the second call the SAS_ZSTRVER, all of the characters in the input string are in the string ABCDE so the routine returns a value of −1.

```
int   i;

i = SAS_ZSTRVER("A B C D E", -1, "ABCD ", -1);
SAS_XPSLOG("SAS_ZSTRVER verifies %d as position (should be 8).", i);
i = SAS_ZSTRVER("A B C D E", -1, "ABCDE ", -1);
SAS_XPSLOG("SAS_ZSTRVER verifies %d as position (should be -1).", i);
```

SAS_ZSWEEP

Sweep a symmetric matrix on a given pivot

Declarations

Type	Variable	Use	Description
double	**matrix*	update	[*order**(*order*+1)/2] matrix to be swept.
int	*row*	input	the row to be swept.
int	*order*	input	the order of the matrix.
char	*flag[order]*	update	the vector to record performed sweepings. The value of *flag*[*order*] is unchanged if the matrix is singular: flag[i]= '0' before the matrix is swept flag[i]= '1' after the matrix is swept.
double	*work[order]*	update	a work vector.
double	*epsilon*	input	the singularity criterion for pivot: epsilon>=0.: x[k,k]<=epsilon. For positive definite matrices (for example, crossproduct matrices) epsilon>=0 should be used. epsilon<0.: fabs(x[k,k])<=−epsilon.

Usage

C	`SAS_ZSWEEP(`*`matrix, row, order, flag, work, epsilon`*`);`
FORTRAN	`CALL SAS_ZSWEEP(`*`matrix, row, order, flag, work, epsilon`*`)`
PL/I	`CALL SAS_ZSWEEP(`*`matrix, row, order, flag, work, epsilon`*`);`

Description

The SAS_ZSWEEP routine sweeps a symmetric matrix on a given pivot. Sweeping performs elementary linear operations like the Gaussian algorithm and should use rational pivoting if possible. If you need to do more than one sweeping, and you can choose the order in which they are done, use the row index, *k*, that corresponds to the largest diagonal element x[*k*,*k*].

SAS_ZSYINF

Return a description of the host system

Declarations

Type	Variable	Use	Description
struct WZ_ENV	**environ*	output	a pointer to a structure describing the environment
long	*rc*	returned	return code 0 = successful

Usage

```
rc = SAS_ZSYINF(environ);
```

Description

SAS_ZSYINF returns the information listed below in the fields in the WZ_ENV structure:

- machine model name (VAX)
- machine model number (11/780)
- machine serial number(s)
- operating system name (VMS)
- operating system family (OS)
- operating system version (5.0)
- job or process id
- SAS Supervisor version number

Refer to the description of the WZ_ENV structure in Chapter 20, "Data Structures."

SAS_ZSYINF *continued*

Example

The following example illustrates how to extract information about the host system where your programming is running:

```
struct WZ_ENV env;

SAS_ZSYINF(&env);
SAS_XPSLOG("Some info returned from SAS_ZSYINF routine.....");
SAS_XPSLOG("First CPU: %s %s %s.",env.model_name,env.model_num,
          env.serial);
SAS_XPSLOG("Operating system: %8b %8b (Family %8b).",
          env.os_name,env.os_num,env.os_family);
SAS_XPSLOG("Jobid is %16b.",env.jobid);
SAS_XPSLOG("SAS System version number: %16b.",env.sup_ver_long);
```

SAS_ZTIME

Convert the current host-specific time to a SAS time value

Declarations

Type	Variable	Use	Description
long	*time*	returned	current time of day as a SAS time value

Usage

```
time = SAS_ZTIME();
```

Description

The SAS_ZTIME routine returns the current time to the second as a SAS time value.

Example

The following example obtains the current time and stores it in a SAS time value. The example also prints the time.

```
long   temp;
char   string[200];

temp = SAS_ZTIME();
SAS_XFXPN(temp, 8, 0, 0, string);
SAS_XPSLOG("Current time is %ld (%8s).", (long)temp, string);
```

SAS_ZTODAY

Obtain today's date as a SAS date value

Declarations

Type	Variable	Use	Description
double	*date*	returned	current date as a SAS date value

Usage

```
date = SAS_ZTODAY();
```

Description

The SAS_ZTODAY routine returns the current date as a SAS date value.

Example

The following example obtains the current date and stores it in a SAS date value. The example also prints the date.

```
double temp;
long   fode;
char   string[200];

SAS_XFNAME("DATE    ", 'F', NULL, &fcode);
temp = SAS_ZTODAY();
SAS_XFXPN(temp, 9, 0, fcode, string);
SAS_XPSLOG("Today's date is %ld (%9s).", (long)temp, string);
```

SAS_ZTREBLD

Locate or add a node to a binary tree

Declarations

Type	Variable	Use	Description
struct TREE	**head-ptr*	update	pointer to the head of the tree. Set this variable to NULL the first time you call the routine.
struct TREE	**now-ptr*	input	pointer to the node to insert or find.
int	*length-type*	input	the size and type of the key field to use for the comparison: >0 = character data of this length 0 = an integer -2 = a double, and it may be a missing value -7 = a nonmissing double.
int	*offset*	input	number of bytes the key field is offset from the beginning of the node.
struct TREE	**return-ptr*	returned	the address of the node found in the tree that matches the node pointed to by *now-ptr*. If *now-ptr* and *return-ptr* are the same, the node did not exist in the tree but was inserted.

Usage

```
C        return-ptr = SAS_ZTREBLD(&head-ptr, now-ptr, length-type, offset);
FORTRAN  return-ptr = SAS_ZTREBLD(IADDR(head-ptr),now-ptr, length-type, offset)
PL/I     return-ptr = SAS_ZTREBLD(ADDR(head-ptr), now-ptr, length-type, offset);
```

Description

The SAS_ZTREBLD routine searches a binary tree to locate a node with a specified key value. If the node does not exist, SAS_ZTREBLD inserts it in the tree.

This routine is implemented using Knuth's method (1973, p. 455).

Example

This example builds an array of nodes and inserts these nodes into a tree. After the tree is built, the SAS_ZTREFND routine is used to search for certain nodes in the tree. Finally, the tree is converted to a linked list and printed.

```
struct NODE {
   struct TREE tree;
   char8 key;
   int value;
   } node[10], testnode;
ptr head;
struct NODE *p;
int i, j;
static char keylist[][9] = {"TOOLKIT ","EXAMPLE ","PROCEDUR",
                            "SORTTEST","TREEING ","KEYLIST ",
                            "MIXED UP","EXAMPLE ","TREEING ",
                            "LAST ONE"};
static int compval[] = {1,5,9,6,2,3,0,4};

   /* fill in the nodes */
for (i=0;i<10;i++) {
   SAS_ZZEROI(node+i, sizeof(struct NODE));
   SAS_ZMOVEI(keylist [i], node[i].key, 8);
   node[i].value = i;
   }

   /* build the tree */
j = U_OFFSET(struct NODE, key);
for (i = 0, head = NULL; i < 10; i++) {
   p = (struct NODE *)SAS_ZTREBLD(&head, node+i, 8, j);
   SAS_XPSLOG("SAS_ZTREBLD new node? %s. (should be %s).",
              reply[p == node+i],reply[i !=7 && i != 8]);
   }

   /* can we find a particular key? */
SAS_ZMOVEI("TREEING ", testnode.key, 8);
p = (struct NODE *)SAS_ZTREFND(&head, &testnode, 8, j);
SAS_XPSLOG("SAS_ZTREFND found match? %s (should be yes).",
          reply[p != NULL]);
if (p != NULL) {
   SAS_XPSLOG("value found=%d (should be 4).", p->value);
   }

   /* try finding a nonexistent key */
SAS_ZMOVEI("NOTTHERE", testnode.key, 8);
p = (struct NODE *)SAS_ZTREFND(&head, &testnode, 8, j);
SAS_XPSLOG("SAS_ZTREFND found match? %s (should be no).",
          reply[p != NULL]);
if (p != NULL) {
   SAS_XPSLOG("(SHOULD NOT SEE THIS) value found=%d.", p->value);
   }
```

SAS_ZTREBLD *continued*

```
   /* convert the tree to a linked list */
SAS_ZTRELST(&head);

   /* print the contents of the linked list */
p = (struct NODE *)head;
for (i=0, p = (struct NODE *)head; p != NULL;
         p = (struct NODE *)(p->tree.llink)) {
   SAS_XPSLOG("Node value is %d (should be %d).",
              p->value,compval[i++]);
   }
```

SAS_ZTREFND

Search a binary tree

Declarations

Type	Variable	Use	Description
struct TREE	**head-ptr*	update	pointer to the head of the tree
struct TREE	**now-ptr*	input	pointer to the node to find
int	*length-type*	input	the size and type of the key field to use for the comparison: >0 = character data of this length 0 = an integer -2 = a double, and it may be a missing value -7 = a nonmissing double
int	*offset*	input	number of bytes the key field is offset from the tree base
struct TREE	**return-ptr*	returned	the address of the node found in the tree that matches the node pointed to by *now-ptr* or NULL if the node is not found

Usage

C	`return-ptr = SAS_ZTREFND(&head-ptr, now-ptr, length-type, offset);`
FORTRAN	`return-ptr = SAS_ZTREFND(IADDR(head-ptr), now-ptr, length-type, offset)`
PL/I	`return-ptr = SAS_ZTREFND(ADDR(head-ptr), now-ptr, length-type, offset);`

Description

The SAS_ZTREFND routine searches a binary tree to locate the node with a specified key value. If you are simply searching for nodes and do not want to add nodes that are not found, it is more efficient to search trees with SAS_ZTREFND than SAS_ZTREBLD.

This routine is implemented using Knuth's method (1973, p. 455).

Example

See the example for the SAS_ZTREBLD routine.

SAS_ZTRELST

Convert a tree built by SAS_ZTREBLD into a linked list

Declarations

Type	Variable	Use	Description
struct TREE	**head-ptr*	update	pointer to the head of the tree

Usage

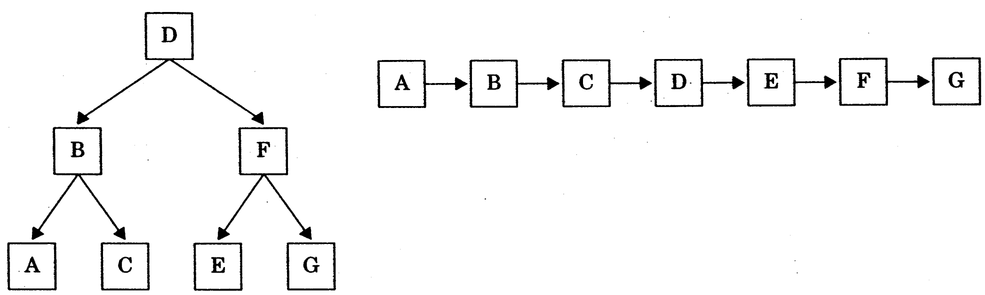
C	`SAS_ZTRELST(&head );`
FORTRAN	`CALL SAS_ZTRELST(IADDR(head))`
PL/I	`CALL SAS_ZTRELST(ADDR(head ));`

Description

The SAS_ZTRELST routine converts a binary tree into a simple list by resetting pointers. Once this routine is called, no more SAS_ZTRE*xxx* routines can be called for this tree because it is now a linked list. Figure 23.3 illustrates this process.

SAS_ZTRELST *continued*

Figure 23.3 *Converting a Tree to a Linked List*



Example

See the example for the SAS_ZTREBLD routine.

SAS_ZVDOTJJ

Compute the dot product of two vectors of possibly different strides

Declarations

Type	Variable	Use	Description
double	**vector-1-ptr*	input	a pointer to the first vector, **v1**
long	*stride-1*	input	the stride of the vector, **v1**
double	**vector-2-ptr*	input	a pointer to the second vector, **v2**
long	*stride-2*	input	the stride for the second vector, **v2**
long	*n*	input	the size of the vectors, **v1** and **v2**
double	*dot-p*	returned	the accumulated dot product

Usage

```
dot-p = SAS_ZVDOTJJ(vector-1-ptr, stride-1, vector-2-ptr, stride-2, n);
```

Description

The SAS_ZVDOTJJ routine computes the dot product of two vectors of possibly different strides. Consider multiplying the matrices **A** and **B**, where **A** is (m,n) and **B** is (n,k). To compute the element (i,j) of the matrix **C**=**AB**, you must compute the dot product of the *i*th row of **A** by the *j*th column of **B**. The stride of a one-dimensional subarray **v** of a multidimensional array **A** is defined to be the distance between two consecutive elements of **v** in storage.

Note that two dimensional arrays are stored as consecutive rows. Consider computing the element $\mathbf{C}_{i,j}$. In **C**, the first element of the *i*th row of **A** is $[n^* \ i]$ with a stride of 1. The first element of the *j*th column of **B** is **B**[*j*] with a stride of *n*.

SAS_ZVXAYJJ

Replace a vector with the sum of the vector and the multiple of another vector of possibly different strides

Declarations

double	**vector-1-ptr*	input	a pointer to the first vector, **v1**
long	*stride-1*	input	the stride of the vector, **v1**
double	**vector-2-ptr*	input	a pointer to the second vector, **v2**
long	*stride-2*	input	the stride for the second vector, **v2**
long	*n*	input	the size of the vectors, **v1** and **v2**
double	*alpha*	input	the scale

Usage

C
```
SAS_ZVXAYJJ(vector-1-ptr, stride-1, vector-2-ptr, stride-2,
      n, alpha);
```
FORTRAN
```
CALL SAS_ZVXAYJJ( vector-1-ptr, stride-1, vector-2-ptr, stride-2,
      n, alpha)
```
PL/I
```
CALL SAS_ZVXAYJJ( vector-1-ptr, stride-1, vector-2-ptr, stride-2,
      n, alpha);
```

Description

The SAS_ZVXAYJJ routine multiplies the contents of *vector-2* by *alpha*, stores that result in *vector-2* and then adds the contents of *vector-2*, to *vector-1*. The result of the addition is stored in *vector-1*.

Let $\mathbf{v}_i$, $i=1,2$ be vectors of size n and stride si respectively. The stride of a one-dimensional subarray **v** of a multidimensional array **A** is the distance between two consecutive elements of **v** in storage. Given a scalar *alpha*, the SAS_ZVXAYJJ routine computes $\mathbf{v}_i$ as follows:

$$\mathbf{v}_1 = \mathbf{v}_1 + alpha\mathbf{v}_2 \quad .$$

SAS_ZVZJJJ

Compute the sum of the vector and the multiple of another vector of possibly different strides and store the result in another vector

Declarations

double	**result-vector-ptr*	output	a pointer to the vector containing the results, **vr**
long	*result-stride*	input	the stride of the result vector, **vr**
double	**vector-1-ptr*	input	a pointer to the first vector, **v1**
long	*stride-1*	input	the stride of the vector, **v1**
double	**vector-2-ptr*	input	a pointer to the second vector, **v2**
long	*stride-2*	input	the stride for the second vector, **v2**
long	*n*	input	the size of the vectors, **v1** and **v2**
double	*alpha*	input	the scale

Usage

C
```
SAS_ZVZJJJ(result-vector-ptr, result-stride, vector-1-ptr,
        stride-1, vector-2-ptr, stride-2, n, alpha);
```
FORTRAN
```
CALL SAS_ZVZJJJ(result-vector-ptr, result-stride, vector-1-ptr,
        stride-1, vector-2-ptr, stride-2, n, alpha)
```
PL/I
```
CALL SAS_ZVZJJJ(result-vector-ptr, result-stride, vector-1-ptr,
        stride-1, vector-2-ptr, stride-2, n, alpha);
```

Description

The SAS_ZVZJJJ routine differs from SAS_ZVXAYJJ in only one way: the result of the multiplied and added vectors is stored in a third vector, not the original vector.

Let $\mathbf{v}_i$, $i=1,2$ be vectors of size n and stride si respectively. The stride of a one-dimensional subarray $\mathbf{v}$ of a multidimensional array $\mathbf{A}$ is the distance between two consecutive elements of $\mathbf{v}$ in storage. Given a scalar *alpha*, the SAS_ZVZJJJ routine computes $\mathbf{r}$ with stride s_r as follows:

$$\mathbf{r} = \mathbf{v}_1 + alpha\mathbf{v}_2$$

SAS_ZYMDJUL

Convert Julian date to year, month, day

Declarations

Type	Variable	Use	Description
double	* *year-ptr*	output	year value
double	**month-ptr*	output	month value
double	**day-ptr*	output	day value
double	**julian-ptr*	input	Julian value
int	*rc*	returned	return code: 0=successful

Usage

```
rc = SAS_ZYMDJUL(year-ptr, month-ptr, day-ptr, julian-ptr);
```

Description

The SAS_ZYMDJUL routine converts Julian dates into year, month, and day values. If the Julian date is missing or is invalid, the year, month, and day values are also set to missing.

Example

See the example for the SAS_ZJULYMD routine.

SAS_ZZEROI

Fill memory with 0s

Declarations

Type	Variable	Use	Description
ptr	*area-ptr*	output	pointer to area to fill with 0s
int	*length*	input	length of area to fill with 0s

SAS_ZZEROI *continued*

Usage

C	`SAS_ZZEROI(area-ptr, length);`
FORTRAN	`CALL SAS_ZZEROI(area-ptr, length)`
PL/I	`CALL SAS_ZZEROI(area-ptr, length);`

Description

The SAS_ZZEROI routine sets a block of memory to 0s.

Example

The following example uses SAS_ZZEROI to set a character string to 0s. The example then uses the $HEX format code to print the hexadecimal value of the string to ensure that it contains all 0s.

```
char  temp[8],
      temp2[16];
long  fcode;

SAS_ZZEROI(temp,8);
SAS_XFNAME("$HEX    ", 'F', NULL, &fcode);
SAS_XFXPC(temp,8,fcode,temp2,16);
SAS_XPSLOG("SAS_ZZEROI set these 8 bytes to %16s in hex
    (should be zeroes).",temp2);
```

References

Fishman, G.S. and Moore, L.R. (1982), "A Statistical Evaluation of Multiplicative Congruential Generators with Modulus $(2^{31}-1)$," *Journal of the American Statistical Association*, 77, 129-136.

Donald Knuth (1973), *The Art of Programming, Volume 3/Sorting and Searching*, Reading, MA: Addison-Wesley, 455.

Chapter 24 Miscellaneous Routines and Values

Introduction

The routines described in this chapter provide special processing needed to write SAS procedures, informats, formats, functions, and CALL routines. In addition, this chapter includes a table of macro variables that permit you to use common host-dependent values, such as the largest floating-point value, without having to program these as literal values.

Note: The routines described in this chapter work with all supported languages except where noted.

Pointer Routines

This group of routines is included with SAS/TOOLKIT software to provide FORTRAN and PL/I programmers with tools for working with pointers. Pointers are not native data types in the FORTRAN language, but they are necessary to be able to interface correctly with SAS/TOOLKIT software. All the routines described in this section can be used with FORTRAN programs.

In addition, although the PL/I language provides a pointer type, it does not permit you to obtain the address of a character literal. You can use the CADDR routine with PL/I programs when you need to pass the address of a character literal.

CADDR

Obtains the address of a character argument

FORTRAN and PL/I languages only

Declarations

Type	Variable	Use	Description
CHAR	*string*	input	the value for which you want an address
INTEGER*4 or PTR	*cptr*	returned	the address of *string*

Usage

```
cptr=CADDR(string)
```

Description

The CADDR function returns the address of a character variable and provides the appropriate descriptive information about the variable. The pointer value returned by this routine enables SAS/TOOLKIT software to handle addresses of character variables appropriately for each operating system. Use the address returned by CADDR when a SAS/TOOLKIT routine requires a pointer to a character (`char *` or `ptr`).

IADDR

Obtains the address of a numeric argument

FORTRAN language only

Declarations

Type	Variable	Use	Description
any numeric	*number*	input	the value for which you want an address
INTEGER*4	*iptr*	returned	the address of *number*

Usage

```
iptr=IADDR(number)
```

Description

The IADDR function provided with SAS/TOOLKIT software returns the address of a numeric variable and stores the address in an INTEGER*4. The address returned by IADDR can be used with a SAS/TOOLKIT routine requiring a pointer to a long integer. The FORTRAN compiler under VMS includes an IADDR function, but other compilers may not. The IADDR function is included with SAS/TOOLKIT software to ensure that this functionality is available on all systems.

LOCPTR

Obtains a pointer to an array element

FORTRAN language only

Declarations

Type	Variable	Use	Description
varies	*base*(0:1)	input	the *base* array used to obtain an index. This array must have the same type as the variable being indexed.
INTEGER*4	*size*	input	the size of the data type declared for *base*. For example, if *base* is an INTEGER*4, *size* is 4.
INTEGER*4	*iptr*	input	the location in memory for which you want to obtain an index.
INTEGER*4	*iloc*	returned	the index of a value in the *base* array.

LOCPTR *continued*

Usage

```
iloc=LOCPTR(base,size,iptr)
```

Description

The LOCPTR routine provides a method for accessing dynamically allocated memory. When you use the SAS_XM routines to allocate memory in a SAS procedure written in the FORTRAN language, you need a method for accessing the contents of that memory. The LOCPTR routine enables you to treat the allocated memory as part of an array that you declare in your program. Any part of the allocated memory can be accessed as an offset to an array.

The LOCPTR routine requires a base array, the size of elements in that array, and the pointer to the allocated memory. You must define the dimensions of the base array to start at 0.

The value returned by LOCPTR, *iloc*, can be used as the index to the base array to access the first element in the array. So, the first element in the allocated memory is BASE(*iloc*) and the *n*th element is BASE($iloc+n-1$).

Note: FORTRAN compilers do not issue warning or error messages when you access elements beyond the range of the specified dimension.

Refer to "Using Dynamically Allocated Memory" in Chapter 26, "Writing Programs in the FORTRAN Language," for a discussion of how to simplify accessing dynamically allocated memory by passing the pointer returned by LOCPTR to a subroutine.

LOCPTR uses the base array passed to it to calculate what element in the array corresponds to the dynamically allocated memory. The index to the base array can be either positive or negative, depending upon where the dynamically allocated memory is in relation to base.

Possible Problems with Using LOCPTR

Although LOCPTR works quite well in most situations, keep in mind these possible problems:

- It is possible that the dynamically allocated memory is so far from the base array that you exceed the limit the FORTRAN compiler imposes on indexes to arrays.
- The indexing method for the base array can be complicated. It requires that you either pass all array elements to subroutines so you can use simpler indexes, or that you calculate the index using the $I+n-1$ algorithm, where I is the value returned by LOCPTR and *n* is the actual number of the array element.
- Most dynamic allocations of memory must be completed before you begin using any of the allocated memory.
- This method may not be completely portable if your operating system limits the dimensions of an array.

Example

Refer to Chapter 26 for more information on LOCPTR and for examples that illustrate how it works.

Format and Informat Definition Routines

This group of routines defines the formats or informats that you create using SAS/TOOLKIT software. These routines are illustrated in Chapter 13, "Writing a SAS Informat," Chapter 14, "Writing a SAS Format," and Chapter 16, "Sample IFFCs in the FORTRAN, PL/I, and IBM 370 Assembler Languages."

FMTDFE

Finishes the definition of formats or informats

Declarations

This routine has no arguments. The value returned by FMTDFE is returned directly to the SAS Supervisor.

Usage

```
return(FMTDFE());
```

Description

The FMTDFE routine signals the end of definitions for formats and informats in a user-written format/informat package. You must call FMTDFE after all calls to FMTDFN are completed. (The FMTDFN routine defines a single format or informat.) When you call FMTDFE, it returns a value needed by the SAS Supervisor so it can access the formats or informats in the program when they are needed.

Always use this routine as shown in "Usage" above.

FMTDFN

Defines a single format or informat

Declarations

Type	Variable	Use	Description
int	*number*	input	number of the routine that processes this format or informat
char	**name*	input	name of the format or informat
int	*min-width*	input	minimum width of the format or informat
int	*max-width*	input	maximum width of the format or informat
int	*default-width*	input	default width of the format or informat
int	*min-decimals*	input	minimum number of decimal places in the format or informat
int	*max-decimals*	input	maximum number of decimal places in the format or informat
int	*default-decimals*	input	default number of decimal places in the format or informat
int	*justification*	input	justification for formatted values 0 = left justification (always use this for informats) 1 = right justification

Usage

```
FMTDFN(number, name, min-width, max-width, default-width,
       min-decimals, max-decimals, default-decimals, justification);
```

Description

The FMTDFN routine describes a format or informat to the SAS Supervisor. You must call FMTDFN for every format or informat included in a format/informat package. Each call to FMTDFN must include all of the arguments described in "Declarations" above. These arguments describe the format or informat to the supervisor so the value passed back to the supervisor by your program is handled correctly.

FMTDFS

Identifies number of formats or informats in a package

Declarations

Type	Variable	Use	Description
int	*number*	input	number of formats or informats defined in a single format/informat package

Usage

```
FMTDFS(number);
```

Description

The FMTDFS routine communicates to the SAS Supervisor how many routines are included in a user-written format/informat package. You must follow the call to FMTDFS with individual calls to FMTDFN for each format or informat.

FMTFNE

Provides address of processing routines

Declarations

This routine has no arguments. The value returned by FMTFNE is returned directly to the SAS Supervisor.

Usage

```
return(FMTFNE());
```

Description

The FMTFNE routine communicates to the SAS Supervisor the location of the routines that process the formats and informats in the package. Always use this routine as illustrated in "Usage" above.

FNCARG

Obtains the value of an argument to a function or CALL routine

Declarations

Type	Variable	Use	Description
int	*arg-number*	input	the number of this argument
int	**type-arg*	output	the type of this argument 1 = numeric 2 = character
ptr	*data-ptr*	output	pointer to the address of the value of the argument
short	**current-len-ptr*	output	pointer to the current length of a character argument
short	**max-len-ptr*	output	pointer to the maximum length of a character argument

Usage

```
FNCARG(arg-number, type-arg, &data-ptr, &current-len-ptr, max-len-ptr);
```

Description

The FNCARG routine returns the value of a single argument. If your function or CALL routine meets all of the following criteria, you do not need to use FNCARG to access the arguments:

- You have a fixed number of arguments for a function or CALL routine.
- You have 10 or fewer arguments.
- None of the arguments are defined to be type 3 (either character or numeric) in the call to FNCDFA.

If all of these conditions are met, you can access arguments by simply referring to the variable names defined in the routine that processes the function or CALL routine. That is, if `rtn1` processes a function that adds two arguments and returns the sum, the definition of `rtn1` might look like the following:

```
int rtn1(value1, value2, return)
double *value1,
       *value2,
       *return;
```

In this case, you can access the arguments as follows:

```
*return = *value1 + *value2;
```

If you are processing a function or CALL routine that permits varying numbers of arguments, the definition of the routine specifies only the required arguments and the return value (for functions). In this case, you must use FNCARG to access the values of all of the optional arguments. You must also use FNCARG to access arguments after the tenth argument for routines that permit that many arguments and to access any argument that permits either numeric or character data. In this case, you must also use FNCARG for all subsequent arguments, even if the number of arguments is fixed.

Notice that the arguments to the FNCARG routine require that you pass pointers and in some cases the addresses of pointers for all of the output arguments. The routine is defined in this way so that you can change the value of the arguments in case you are writing a CALL routine. For a numeric argument, you can change the value of the argument by changing the data pointed to by *data-ptr*. For a character argument, you can also change the length pointed to by *current-len-ptr*. You cannot, however, change the maximum length, so if you change a character argument, be sure that the new length of the argument does not exceed the value pointed to by *max-len-ptr*.

Be sure that the value you specify for *arg-number* does not exceed the number of arguments used with the function or CALL routine. If your function or CALL routine permits a varying number of arguments, use the FNCN routine to determine how many arguments the user specified.

FNCDFA

Describes the arguments to a function or CALL routine

Declarations

Type	Variable	Use	Description
int	*routine-number*	input	the number of the routine using this argument
int	*arg-number*	input	the number of this argument
int	*type-arg*	input	the type of this argument 1 = numeric 2 = character 3 = either numeric or character

Usage

```
FNCDFA(routine-number, arg-number, type-arg);
```

Description

The FNCDFA routine describes a single argument for a function or CALL routine. You must call FNCDFA for every argument that can be used in a function or CALL routine up to the tenth argument. The calls to FNCDFA follow the call to FNCDFN for the routine. You must call FNCDFA for the *max-args* number of arguments specified in the call to FNCDFN, if *max-args* is 10 or less. If the

FNCDFA *continued*

max-args argument to FNCDFN is greater than 10 or if it is −1 (indicating any number of arguments is permitted), call FNCDFA only 10 times. All arguments after the tenth one must have the same attributes as the tenth one.

Note: To permit arguments after the tenth argument to have different types, specify a type 3 for the tenth argument, even if you explicitly require numeric or character values. You can test whether the correct type of data is supplied when you call FNCARG to retrieve the value of the argument.

FNCDFE

Finishes the definition of functions or CALL routines

Declarations

This routine has no arguments. The value returned by FNCDFE is returned directly to the SAS Supervisor.

Usage

```
return(FNCDFE());
```

Description

The FNCDFE routine signals the end of definitions for functions and CALL routines in a user-written function/CALL routine package. You must call FNCDFE after all calls to FNCDFN are completed. (The FNCDFN routine defines a single function or CALL routine.) When you call FNCDFE, it returns a value needed by the SAS Supervisor so it can access the functions or CALL routines in the program when they are needed.

Always use this routine as shown in "Usage" above.

FNCDFN

Defines a single function or CALL routine

Declarations

Type	Variable	Use	Description
int	*number*	input	number of the routine that processes this function or CALL routine
char	**name*	input	name of the function or CALL routine
int	*min-args*	input	minimum number of arguments for the function or CALL routine
int	*max-args*	input	maximum number of arguments for the function or CALL routine −1 = any number of arguments is acceptable
int	*type-returned*	input	the type of value returned 0 = no return value (indicates a CALL routine) 1 = numeric 2 = character

Usage

```
FNCDFN(number, name, min-args, max-args, type-returned);
```

Description

The FNCDFN routine describes a function or CALL routine to the SAS Supervisor. You must call FNCDFN for every function or CALL routine included in a function/CALL routine package. Each call to FNCDFN must include all of the arguments described in "Declarations." These arguments describe the function or CALL routine to the supervisor so the supervisor can ensure that the function or CALL routine is called correctly before it passes the values to your routine. After calling FNCDFN, you must call FNCDFA to describe each argument for the function or CALL routine.

FNCDFS

Identifies number of functions or CALL routines in a package

Declarations

Type	Variable	Use	Description
int	*number*	input	number of functions or CALL routines defined in a single function/CALL routine package

Usage

```
FNCDFS(number);
```

Description

The FNCDFS routine communicates to the SAS Supervisor how many routines are included in a user-written function/CALL routine package. You must follow the call to FNCDFS with individual calls to FNCDFN for each function or CALL routine.

FNCFNE

Provides address of processing routines

Declarations

This routine has no arguments. The value returned by FNCFNE is returned directly to the SAS Supervisor.

Usage

```
return(FNCFNE());
```

Description

The FNCFNE routine communicates to the SAS Supervisor the location of the routines that process the functions and CALL routines in the package. Always use this routine as illustrated in "Usage" above.

FNCN

Determines how many arguments were used with a function or CALL routine

Declarations

Type	Variable	Use	Description
int	*number*	output	number of arguments specified by the user in the call to the function or CALL routine

Usage

```
FNCN(&number);
```

Description

The FNCN routine returns the number of arguments specified by the user in the call to the function or CALL routine. If your function or CALL routine permits a varying number of arguments, call FNCN to find out how many arguments were used, and then call FNCARG to access the value of each argument. Be sure that the number of calls to FNCARG does not exceed the value of *number* returned by FNCN.

SF Routines

The SF routines provide access to the information stored in the statement structures by the parser. Your program calls SAS_XSPARSE to parse the statements specified by the user when the procedure is invoked. The bit arguments, parameters, data set references, statement lists, and other grammar elements are stored in a series of statement structures using the instructions you provided in your grammar. Before your procedure can process the statements specified by the user, it must obtain the information stored in the statement structures. Use the routines listed in Table 24.1 to access the statement structures.

Table 24.1 *Summary of SF Routines*

SF Routine	Description
SFC8	accesses the value of a character parameter
SFC8D	accesses the value of a character parameter or uses a default
SFF	accesses the value of a numeric parameter
SFFD	accesses the value of a numeric parameter or uses a default

(continued)

Table 24.1
(continued)

SF Routine	Description
SFFILE	accesses a file pointer
SFLD	accesses a specific statement element
SFLP	accesses a list
SFMODE	determines the mode of an element
SFN	determines the number of items in a list
SFNEXT	accesses the next statement structure
SFNFLD	determines the number of lists, parameters, and data set references
SFOPT	accesses the value of a bit argument
SFTYPE	determines the type of an element. This routine also indicates if the user has specified the statement or parameter that corresponds to this element of the statement structure.

Note: All of these routines use an argument called *pointer* in the "Declarations" and "Usage" sections of the following sections. If the information you need to access is stored in the procedure's statement structure (which is always the first statement structure in a chain), use `proc.head` for the value of *pointer*. Otherwise, use the value returned by a call to SFNEXT for *pointer*. In addition, many of these routines use *field* as one of the arguments. For this argument, use the same number that you specified in the grammar for the corresponding grammar element. For example, if you coded the following actions to define a file and a parameter, you must use 3 for the value of *field* in the call to the SFFILE routine (to access the data set) and 5 for the value of *field* in the call to SFF (to access the parameter):

```
@DS(3,1 . . . )
@PARM(5,1)
```

SFC8

Accesses the value of a character parameter

Declarations

Type	Variable	Use	Description
ptr	*pointer*	input	pointer to the statement structure.
int	*field*	input	field number. This is the same number specified in the grammar for the parameter.
ptr	*parmval*	returned	the 8-byte parameter value.

Usage

```
parmval = SFC8(pointer, field);
```

Description

The SFC8 routine returns the value of an 8-byte character parameter specified by the user. Before you call this routine, call SFTYPE and test for a nonzero return value to be sure that the user specified this character parameter.

SFC8D

Accesses the value of a character parameter or uses a default

Declarations

Type	Variable	Use	Description
ptr	*pointer*	input	pointer to the statement structure.
int	*field*	input	field number. This is the same number specified in the grammar for the parameter.
ptr	*default-ptr*	input	default value if the user did not specify this parameter.
ptr	*parmval-ptr*	returned	the 8-byte parameter value.

Usage

```
parmval-ptr = SFC8D(pointer, field, default-ptr);
```

SFC8D *continued*

Description

The SFC8D routine returns the value of an 8-byte character parameter specified by the user. If the user omits the parameter, SFC8D returns the default value pointed to by *default-ptr*.

SFF

Accesses the value of a numeric parameter

Declarations

Type	Variable	Use	Description
ptr	*pointer*	input	pointer to the statement structure.
int	*field*	input	field number. This is the same number specified in the grammar for the parameter.
double	*parmval*	returned	the value of the numeric parameter.

Usage

```
parmval = SFF(pointer, field);
```

Description

The SFF routine returns the value of a numeric parameter specified by the user. Before you call this routine, call SFTYPE and test for a nonzero return value to be sure that the user specified this numeric parameter.

SFFD

Accesses the value of a numeric parameter or uses a default

Declarations

Type	**Variable**	**Use**	**Description**
ptr	*pointer*	input	pointer to the statement structure.
int	*field*	input	field number. This is the same number specified in the grammar for the parameter
double	*default*	input	default value if the user did not specify this parameter.
double	*parmval*	returned	the value of the numeric parameter.

Usage

```
parmval = SFFD(pointer, field, default);
```

Description

The SFFD routine returns the value of a numeric parameter specified by the user. If the user omits the parameter, SFFD returns a default value, *default*.

SFFILE

Accesses a file pointer

Declarations

Type	**Variable**	**Use**	**Description**
ptr	*pointer*	input	pointer to the statement structure.
int	*field*	input	field number. This is the same number specified in the grammar for the data set reference.
ptr	*file-ptr*	returned	a pointer to a fileid.

Usage

```
file-ptr = SFFILE(pointer, field);
```

SFFILE *continued*

Description

The SFFILE routine returns a pointer to a data set named by the user. If you have included the @DSDFLT semantic action in your grammar to supply a default data set, SFFILE returns a pointer to the appropriate default data set when the user omits a data set reference. You use the value returned by SFFILE in calls to SAS_XBY, SAS_XO, and some SAS_XV routines.

SFLD

Accesses a specific statement element

Declarations

Type	Variable	Use	Description
ptr	*pointer*	input	pointer to the statement structure.
int	*field*	input	field number. This is the same number specified in the grammar for the parameter, data set reference, list, or other semantic action that creates the element.
ptr	*element-ptr*	returned	a pointer to a statement structure element. **Note:** You must cast the value returned by SFLD to `struct STLIST *`.

Usage

```
element-ptr = SFLD(pointer, field);
```

Description

The SFLD routine returns a pointer to a specific STLIST structure in the statement structure. The parser creates an STLIST structure for each parameter, data set reference, or list that is permitted by the grammar for your procedure. In most cases, you do not need to call SFLD to access the STLIST structure because other SF routines provide the precise information you want to access. For example, to obtain a character parameter value, use SFC8 or SFC8D. For numeric parameter values, use SFF or SFFD. For data set references use SFFILE. For lists, use SFLP.

In at least one situation, you need the information returned by SFLD instead of the values returned by the routines described in the previous paragraph. The following example shows a call to SAS_XVDFLST to create a default list. In this case, you want to omit the variables listed in the BY statement from the default list of variables. The BY statement is always stored in the first STLIST structure (indexed by 0) of the procedure's statement structure. You need the pointer returned by the call to SFLD for both the default list you are building and for the list you want to omit from the default list in the call to SAS_XVDFLST. (This is a different pointer than the list pointer you use in the SAS_XVGET routines.)

```
ptr    infile;

struct STLIST *varlist,
              *bylist;

      /* get file pointer */
   infile = SFFILE(proc.head, 1);

      /* access VAR list */
   varlist = (struct STLIST *)SFLD(proc.head, 4);

      /* build default VAR list if none specified */
   bylist = (struct STLIST *)SFLD(proc.head, 0);
   SAS_XVDFLST(infile, 1, varlist, 1, bylist);
```

SFLP

Accesses a list

Declarations

Type	Variable	Use	Description
ptr	*pointer*	input	pointer to the statement structure.
int	*field*	input	field number. This is the same number specified in the grammar for the list.
ptr	*list-ptr*	returned	a pointer to a list of items.

Usage

```
list-ptr = SFLP(pointer, field);
```

Description

The SFLP routine returns a pointer to a list of values. Before you call this routine, call SFTYPE and test for a nonzero return value to be sure that the user specified this list.

The type of values in the list depends on the semantic action that created the list. For example, one of the most common lists is the variable list. In this case, the grammar uses the @STMTLIST semantic action. When the parser processes this element, it creates a list of short integers that indicate which variables from the data set variable list were specified in the statement. Note that a variable list does not create a list of variable names or values. It consists of a list of variable numbers. The pointer to a list of variable numbers is used in calls to the SAS_XV routines.

SFLP *continued*

You can store other lists in the statement structure, such as lists of numeric values or SAS names (8-byte character strings).

To use the `ptr` returned by SFLP, be sure to cast it to the appropriate pointer type, as shown in Table 24.2.

Table 24.2
Pointer Types for Lists

Pointer Type	Contents of List
`char *`	list of character strings.
`short *`	list of variable numbers.
`double *`	list of numeric values. Even a list of integers is stored in this type of array.
`char *`	list of 17-byte character values. This type of list can contain *libref.member-name* entries.
`char8 *`	list of SAS names.

SFMODE

Determines the mode of an element

Declarations

Type	Variable	Use	Description
ptr	*pointer*	input	pointer to the statement structure.
int	*field*	input	field number. This is the same number specified in the grammar for the parameter, list, or data set reference.
int	*mode*	returned	the mode associated with a specific statement structure element.

Usage

```
mode = SFMODE(pointer, field);
```

Description

The SFMODE routine returns the mode of an element. In combination with the type, returned by SFTYPE, the mode indicates what kind of information is stored in an STLIST structure. The type indicates if it is a parameter, a list, or a data set reference. The mode further qualifies this information by indicating whether the data are numeric or character, a variable or value list, or an input or output data set.

Refer to Table 20.2 in Chapter 20, "Data Structures," for more information on the modes and types stored in the statement structures.

SFN

Determines the number of items in a list

Declarations

Type	Variable	Use	Description
ptr	*pointer*	input	pointer to the statement structure.
int	*field*	input	field number. This is the same number specified in the grammar for the parameter.
int	*number*	returned	the number of items in a list.

Usage

```
number = SFN(pointer, field);
```

Description

The SFN routine returns the number of items in a list. If SFN returns 0, the user did not specify the syntax element that creates the list.

For variable lists, the number returned by SFN is useful for determining how many calls you need to make to the SAS_XVGET and SAS_XVPUT routines.

SFNEXT

Accesses the next statement structure

Declarations

Type	Variable	Use	Description
ptr	*pointer*	input	pointer to the statement structure.
ptr	*next-ptr*	returned	pointer to the next statement structure in a chain of structures.

Usage

```
next-ptr = SFNEXT(pointer);
```

SFNEXT *continued*

Description

The SFNEXT routine returns a pointer to the next statement structure in a chain of structures. Each time your grammar uses a STMTINIT semantic action, the parser creates a new statement structure and chains it to the previous structure. The first structure in the chain is the one that contains information for the PROC statement. You can also store information for other statements in the procedure's statement structure. The procedure's statement structure is pointed to by the `head` field in the PROC structure. To access the first statement structure, simply use the `proc.head` pointer. The following example shows how to access the second and third structure:

```
stmtptr   second,
          third;

second = SFNEXT(proc.head);
third = SFNEXT(second);
```

To access the information stored in the second and later statement structures, use the pointers to these structures instead of `proc.head` in calls to the other SF routines.

SFNFLD

Determines the number of lists, parameters, and data set references

Declarations

Type	Variable	Use	Description
ptr	*pointer*	input	pointer to the statement structure.
int	*nfld*	returned	the number of STLIST structures allocated for a statement structure.

Usage

```
nfld = SFNFLD(pointer);
```

Description

The SFNFLD routine returns the largest number assigned to a list, parameter, and data set reference by your grammar. You can use this number to create a loop that processes all of the elements in a statement structure. First obtain the maximum number of elements by calling SFNFLD, then call SFTYPE and SFMODE to determine the kind of information stored in the element, and finally call the appropriate routine to access the information, such as SFFILE, SFF, SFC8, or SFLP. The following example illustrates this sequence:

```
stmtptr stmt_ptr;
int i;

for (i = 0; i < (SFNFLD(stmt_ptr)); i++) {
   if (SFTYPE(stmt_ptr, i) == 4)
      call SFFILE for a data set reference
   else if (SFTYPE(stmt_ptr,i) == 3)
      call SFLP for a list
   else if ((SFTYPE(stmt_ptr, i) == 2) && (SFMODE(stmt_ptr, i) == 1))
      call SFF for a numeric parameter
   else if ((SFTYPE(stmtptr, i) == 2) && (SFMODE(stmtptr, i) == 3))
      call SFC8 for an extra-long character parameter
}
```

SFOPT

Accesses the value of a bit argument

Declarations

Type	Variable	Use	Description
ptr	*pointer*	input	pointer to the statement structure.
int	*option*	input	option number in the range of 0 to 63. This is the same number specified in the grammar for the option.
int	*setting*	returned	the value of the option 0 = argument not specified 1 = argument specified by user.

Usage

```
setting = SFOPT(pointer, option);
```

Description

The SFOPT routine returns the setting of a bit argument. If the user specifies the argument, SFOPT returns a 1; otherwise it returns a 0.

SFTYPE

Determines the type of an element

Declarations

Type	Variable	Use	Description
ptr	*pointer*	input	pointer to the statement structure.
int	*field*	input	field number. This is the same number specified in the grammar for the parameter.
int	*type*	returned	the type associated with a specific statement structure element. A nonzero value indicates that the user specified the grammar element associated with this element of the statement structure.

Usage

```
type = SFTYPE(pointer, field);
```

Description

The SFTYPE routine returns the type of an element. In combination with the mode, returned by SFMODE, the type indicates what kind of information is stored in an STLIST structure. The type indicates if it is a parameter, a list, or a data set reference. The mode further qualifies this information by indicating whether the data are numeric or character, a variable or value list, or an input or output data set.

Refer to Table 20.2 in Chapter 20 for more information on the modes and types stored in the statement structures.

Interface Routines

The UWP routines set up the SAS/TOOLKIT environment for a procedure. These routines are also needed by informats, formats, functions, and CALL routines that use SAS_X or SAS_Z routines.

UWPRCC

Sets up interface between a C or assembler program and the SAS/TOOLKIT environment

C and IBM 370 assembler languages only

Declarations

This routine passes either the address of the PROC structure or 0.

Usage

```
UWPRCC(&proc);  /* for procedures */
UWPRCC(0);      /* for IFFCs */
```

Description

The UWPRCC routine sets up the interface between a C or IBM 370 assembler procedure and the SAS/TOOLKIT environment. You must call UWPRCC as the first statement in a procedure. In addition, if you want to use any of the SAS_X or SAS_Z routines in an informat, format, function, or CALL routine, you must call UWPRCC as the first statement in the IFFMAI routine.

UWPRCF

Sets up interface between a FORTRAN program and the SAS/TOOLKIT environment

FORTRAN language only

Declarations

This routine passes either the address of the PROC structure or 0.

Usage

```
CALL UWPRCF(IADDR(proc))  /* for procedures */
CALL UWPRCF(0)            /* for IFFCs */
```

Description

The UWPRCF routine sets up the interface between a FORTRAN program and the SAS/TOOLKIT environment. You must call UWPRCF as the first statement in a procedure. In addition, if you want to use any of the SAS_X or SAS_Z routines in an informat, format, function, or CALL routine, you must call UWPRCF as the first statement in the IFFMAI routine.

UWPRCP

Sets up interface between a PL/I program and the SAS/TOOLKIT environment

PL/I language only

Declarations

This routine passes either the address of the PROC structure or 0.

Usage

```
CALL UWPRCP(ADDR(proc));  /* for procedures */
CALL UWPRCP(0);           /* for IFFCs */
```

Description

The UWPRCP routine sets up the interface between a PL/I procedure and the SAS/TOOLKIT environment. You must call UWPRCP as the first statement in a procedure. In addition, if you want to use any of the SAS_X or SAS_Z routines in an informat, format, function, or CALL routine, you must call UWPRCP as the first statement in the IFFMAI routine.

Routines and Macro Variables to Replace Host-Dependent Constants

This section describes the SAS_NUL routine and provides a table of macro variables you can use instead of including host-dependent constant values in your program.

Macro Variables

Using the MAC*xxxx* variables listed in Table 24.3 helps to make your program portable.

Table 24.3
Macro Variables

Macro Variable	Description
Common	
MACBIG	largest double-precision floating-point value
MACFBIG	largest floating-point value
MACFMISSING	SAS missing value cast as a `float`
MACFSMALL	smallest floating-point value

(continued)

Table 24.3
(continued)

Macro Variable	Description
MACINT	largest integer (the native `int`)
MACLONG	largest long integer
MACMISSING	SAS missing value as a double-precision, floating-point value
MACNULL	NULL value
MACSHORT	largest short integer
MACSMALL	smallest double-precision, floating-point value
Less Common	
MACADDBIG	MACBIG/2
MACBITS	number of bits per byte
MACCUBEBIG	cube root of MACBIG
MACCUBESMALL	cube root of MACSMALL
MACEPS	1+MACEPS=1
MACLOGBIG	natural log of MACBIG
MACLOGLONG	natural log of MACLONG
MACLOGSMALL	natural log of MACSMALL
MACMAXDBLLNG	maximum number of bytes for a double-precision, floating-point value
MACMINDBLLNG	minimum number of bytes for a double-precision, floating-point value
MACSQRTBIG	square root of MACBIG
MACSQRTEPS	square root of MACEPS
MACSQRTSMALL	square root of MACSMALL
MACSSCPBIG	MACSQRTBIG/100
MACSUMBIG	MACBIG/1000
MACSWCPBIG	cube root of MACSUMBIG
MACTENBIG	maximum decimal exponent
MACTENSMALL	minimum decimal exponent

SAS_NUL

Provides a NULL value that can be used with SAS/TOOLKIT software

PL/I language only

Declarations

Type	Variable	Use	Description
ptr	*pointer*	returned	a pointer that can be set to NULL

Usage

```
pointer = SASNUL();
```

Description

The SASNUL routine returns a pointer value set to NULL. This NULL value can be used with SAS/TOOLKIT functions that require NULL as an argument. Use this routine instead of the PL/I NULL built-in routine, because that routine does not return a true NULL value under all operating systems.

Part 7

Language Dependencies

Chapter 25 Writing Programs in the C Language

Introduction

Many software applications must be written so that the same program can be compiled and linked under more than one operating system. Developing programs that will compile and run successfully under many systems, even if your current environment does not require portability, prevents the need to recode programs if your environment changes. This chapter describes some of the requirements for writing portable code. The first half of the chapter contains information on good coding practices for writing programs in C, which is especially helpful for programmers who are new to this language. The second half of the chapter discusses coding techniques to avoid to make your programs portable.

Important Note on Null Terminators

In C, you can ususally indicate the end of a character string with a null terminator. In many of the functions provided with SAS/TOOLKIT software, you cannot assume that character strings are null terminated. Because SAS/TOOLKIT software is designed to work with languages other than C, it does not depend on null terminators to signal the end of character strings. If a null-terminated character string is accepted by a function, the description of the function says so; otherwise, you must indicate the length of a character string.

Good C Coding Standards

The following sections discuss good coding standards. Although the techniques described here are not necessary for portable code, they make C programs clearer and more concise.

Case

Uppercase all macros in the `#define` statement except those that redefine a standard C subroutine (for example, `getc`). All structure tags should be in uppercase.

Code Blocks

Always indent a subordinate code block, marked by {. In addition, place the opening and closing brackets in the same location for each code block. There are several acceptable methods. The following examples show two acceptable forms:

```
if (x == 1) {
   y = 1;
   z = 3;
}
```

```
if (x == 1)
   {
    y = 1;
    z = 3;
   }
```

It is also a good practice to put braces around any `for` or `while` loop, even if the block has only one statement:

```
for (i = 0; i < j; i++) {
   x += k[i];
}
```

The reason for this practice is that you can insert new statements into the loop by placing them within the existing braces. Thus, you do not need to remember to add braces as well as the new statements. For example, if you omit the braces from the `for` loop, as shown on the left in the following example, and then later want to add another statement to the loop, you are likely to add the line as shown here on the right. The last line in the example on the right looks like it is part of the `for` loop because of the indention, but in fact, it is executed after the `for` loop is finished.

Acceptable

```
   /* single statement in loop */
for (i = 0; i < j; i++)
   x += k[i];
```

Incorrect

```
   /* adding another statement */
for (i = 0; i < j; i++)
   x += k[i];
   y += x;   /* not in loop */
```

If you include the braces in the original, as shown here on the left, you can insert the line more easily, as in the example on the right.

Better

```
   /* single statement in loop */
for (i = 0; i < j; i++) {
   x += k[i];
}
```

Correct

```
   /* adding a statement in braces */
for (i = 0; i < j; i++) {
   x += k[i];
   y += x;
}
```

Making Your Programs More Readable

It is a good practice to put only one statement per line to make your code more readable. Also, use blank lines to separate code sections, and comment your code to explain and separate code sections.

In particular, remember to comment code segments that might look like mistakes or that are difficult to understand. For example, this section of code intentionally omits a `break` statement at the end of the first case. The comment notes that the omission is not a mistake:

```
select(x) {
   case 1:
      y = 2;    /* intentional fall-through */
```

```
case 2:
   z = 3;
   break;
case 3:
   q = 4;
   break;
}
```

Some other situations that may look like mistakes are illustrated here:

```
if (x = y)      /* intentional assignment and zero comparison */
   z = 2;

if (a & b)      /* single ampersand intentional */
```

Compiler Warnings

If the compiler issues warnings, eliminate the problems that caused the warning. A common warning is `variable x not used`. This warning usually means that either you misspelled the variable name when you used it, or you accidentally commented out the section of code that uses it, or you no longer are using a variable you once needed in the program. Any of these conditions is an error and should be corrected. Of course, the warning can also occur when you intended to use a variable and then decided not to. In this case, the warning does not cause problems, but it is better to remove the variable definition to reduce memory usage and to eliminate unnecessary warnings that may obscure other problems.

Native C Coding

If you have recently learned the C language and have previously used other languages such as PL/I and FORTRAN, you may find some of the concepts of C a little unusual at first. For example, you may be unfamiliar with 0-based arrays and braces instead of BEGIN (or DO) and END statements.

Although it is possible to use the C preprocessor to hide many of the native C features, this is not recommended. Redefining elements of the language makes your program far less readable. The following list provides some examples of native C programming techniques that you need to be familiar with:

- 0-based arrays
- braces instead of BEGIN (or DO) and END statements
- `while` at the bottom of a DO loop
- the `continue` and `break` statements
- the ternary operator (? :)
- the `switch/case` statements, which are similar to the SELECT/WHEN statements of PL/I.

For basic information on these and other C programming techniques, refer to *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie.

Techniques for Writing Portable Programs

The C language was designed with portability in mind, but writing programs in C does not automatically make them portable. The C language provides many different constructs that are implemented differently on different operating systems. This section describes C programming techniques that are not portably implemented.

Naming Conventions

This section describes recommended coding techniques to avoid operating system dependencies in the following areas:

- variable names
- subroutine names
- `#include` files.

Variable Names

Use only the standard alphanumeric characters, plus the underscore. Some C compilers may permit other characters, but not all do. Do not use the underscore as the first character of the name; some compilers reserve the leading underscore to indicate special usage.

Subroutine Names

Limit the names of external routines to seven characters. Some operating systems, such as MVS and CMS, are limited to eight characters for external names. Some compilers, such as SAS/C, use an extra character to define additional CSECTs for the module. For example, the external routine `abcdefg` has CSECTs for ABCDEFG$, ABCDEFG:, and so on. If an external name is over seven characters, the SAS/C Compiler truncates it. If you have multiple external names of more than seven characters that differ only after the first seven characters, you may have names that are defined more than once when you link the program. Limiting names to seven characters eliminates this problem.

Note also that some systems, such as CMS and MVS hosts, ignore the casing of external names. This means, for example, that a routine named `AxBy` has the same name as `AXBY` or `axby`. Ensure that external subroutine names are unique irrespective of casing. This restriction does not apply to internal (that is, `static`) subroutines.

`#include` Files

The names of `#include` files should follow the same rules as those for variable names and should be no more than eight characters in length, not including the `.h` portion. For example, the name in the example on the left in the following example is acceptable on all operating systems. Under MVS and CMS, the name in the example on the right is too long and would be truncated to the same name shown on the left:

Portable

```
   /* acceptable */
#include "abcdefgh.h"
```

Not Portable

```
   /* too long */
#include "abcdefghi.h"
```

Data Types

This section describes recommended coding techniques to avoid operating system dependencies in the following areas:

- signed and unsigned characters
- data sizes
- character arrays
- using long or short integers
- character constants
- integers and pointers.

Signed and Unsigned Characters

A `char` is either signed or unsigned, depending on the compiler implementation. If you declare a `char` variable and you want to compare it to values less than 0, explicitly declare it as `signed char`. To compare to values greater than 127, declare the variable as `unsigned char`.

Data Sizes

C compilers assume different sizes for data types when the size is not specifically declared. Never assume the length of an `int`, `short`, `long`, `double`, or any structure. For example, on CMS and MVS hosts, an `int` is 4 bytes, but on PCs, an `int` is 2 bytes. To avoid problems, do not assume the size of a data type; use the `sizeof` function to determine the appropriate size.

Note: You can safely assume that the size of a `char` is 1.

To determine the size of structures, do not simply add the sizes of the different elements together. Use the `sizeof` function on the entire structure to determine its size. Some compilers perform extensive alignment of the elements within a structure, so the structure may actually require more space than the combined number of bytes. In the following example, some compilers add padding after the `int` and `char`:

```
struct ABC {
   int a;          /* possible 4-byte pad */
   double b;
   char c;         /* possible 3-byte pad */
   char *d;
};
```

Character Arrays

Do not assume that two character elements in an array are contiguous. For example, the following declarations are not equivalent:

```
struct ABC {
   char x1[5];
   char x2[6];
};

struct ABC {
   char xxx[11];
};
```

Using Long or Short Integers

When you declare an integer, be sure to declare it as `long` if there is any chance that the value of the integer will ever exceed 32,767, the maximum value of a signed `short`. Keep in mind that on some systems, an `int` is actually a `short`.

Character Constants

The C language treats the following values differently:

```
'A'
"A"
```

The first value is a single character and is treated as an integer value. The second value is a character string with one character, and it is handled as a pointer to a character.

When you are using character constants, only a single character value can be enclosed in single quotation marks, as shown in the following example:

```
abc('Z');
```

Some compilers allow longer strings but many do not.

If you have a character constant with more than one character, use double quotation marks, as follows:

```
abc("XYZ");
```

When a single character is passed as an argument, it is promoted to an `int`. The value received by the routine is different depending upon the host system running the program; that is, on ASCII-based systems the value passed is different than the value passed on an EBCDIC-based system. To avoid host dependencies, do not test the hexadecimal value of a character literal in a called routine. For example, in the following `abc` routine, the first test of `value` works on all systems. The second test works only on ASCII-based systems, and the third test works only on EBCDIC systems.

```
abc('X');

void abc(value)
   int value;
   {
   if (value == 'X')    /* portable */

   if (value == 0x58)   /* only for ASCII */

   if (value == 0xE7)   /* only for EBCDIC */
   }
```

Integers and Pointers

On most but not all systems, a long integer and a pointer are the same size. Do not assume that these two types are the same size. Also, do not overlay integers and pointers and then perform arithmetic operations with the integers. The C language allows you to add and subtract pointers, so it is not necessary to overlay numerics. Attempting to overlay pointers and integers causes problems on some systems.

Using Data

This section describes recommended coding techniques to avoid operating system dependencies in the following areas:

- storage of numbers
- initializing variable values
- modifying character literals
- casting pointers
- structures as function arguments
- use of & operators on non-lvalues
- null pointers.

Storage of Numbers

The actual method of storing numbers in memory varies per host. These differences apply to both integers and floating-point values. For example, on CMS and MVS hosts, a long integer with the value of 1 is stored in memory as 00000001 (in hex). Under VMS, the value of 1 is stored as 01000000, with the least significant byte on the left. Therefore, avoid coding your program based on assumptions about how numbers are stored.

Note: Shift operations on nonnegative numbers work consistently, so the following statements work correctly on all systems:

```
x = 1;
x <<= 8;
```

The result of this operation under CMS and MVS is stored as 00000100 and as 00010000 under VMS. It is important to note, however, that shifting negative numbers does not produce consistent results on all compilers so avoid such shifting.

Initializing Variable Values

Some compilers do not permit you to initialize a variable's value unless you use the `static` attribute. The following examples illustrate declarations that are not portable and the corresponding portable statements:

Not Portable	Portable
`char[x]= {'A','B','C'};`	`static char[x]= {'A','B','C'};`
`short a = 5;`	`static short a = 5;`

Modifying Character Literals

Never modify character literals or static character strings or pass them to a routine that modifies the argument. On some systems, character literals are stored in the program's read-only memory area. On other systems, changing a character literal value in a function can change another part of the program causing unpredictable results. The following example illustrates several problems with overwriting character literals:

```
static char x[] = {'A','B','C'};

memcpy(x, "XYZ", 3);        /* tries to overwrite x */
memcpy("XYZ", x, 3);        /* tries to overwrite the literal */
```

The first call to `memcpy` attempts to overwrite the static character string, `x`, which could cause the program to abend if the value is stored in read-only memory. The second call to `memcpy` attempts to change the literal XYZ which can cause problems if XYZ is used later in the program as well. For example, if you followed the statements shown in the preceding example with a later comparison to XYZ, the stored value of XYZ may be different than what it appears to be. In this case, `q` equals XYZ, but XYZ now has a different value:

```
static char q[] = {'X','Y','Z'};

if (memcmp(q, "XYZ", 3) == 0)
```

Do not pass a character literal to a function that updates the value passed to it. The `abc` function illustrated changes the value passed to it, so the statements that call `abc` should not pass either literal values or pointers to static values.

```
static char x[] = {'X','Y','Z'};

abc(x);                    /* x will be overwritten */
abc("XYZ");                /* the literal will be overwritten */

void abc(value)
   char *value;
   {
   if (value[0] == 'X')
      value[0] = 'Q';      /* overwrites byte 0 */
```

Casting Pointers

You should limit casting pointers to one of two situations:

- casting back and forth from a `char *`
- obtaining memory from a memory allocation routine, such as SAS_XMEMGET. When you use the SAS_XMEMGET routine to obtain memory, you need to cast the pointer returned by SAS_XMEMGET to the appropriate pointer type for the data you will store in that memory.

```
struct ABC *x;

x = (struct ABC *) SAS_XMEMGET(sizeof(struct ABC));
```

Casting pointers between two dissimilar types is not advisable. For example, if you attempt to cast a pointer-to-char to a pointer-to-int, you can lose necessary information about the addressing of the character.

Structures as Function Arguments

Do not pass structures as arguments to functions; instead, pass a pointer to the structure. Passing structures requires much more overhead at execution time, and some compilers may not pass structures as expected.

Use of & Operators on Non-Lvalues

Recall that an lvalue is an expression that can appear on the left side of an equal sign in an assignment statement. Do not use the unary operator (&) on any non-lvalue. For example, in the following statements, `x` is not an lvalue because it cannot be assigned a value by a simple assignment statement. Because it is not an lvalue, some compilers do not permit you to pass the address of `x`, as shown in the first call to `abc`. Instead, use the forms shown in either the second or third call to `abc`.

```
char x[5];

x = NULL;    /* the C language does not permit this */
abc(&x);     /* some compilers reject this */
abc(x);      /* this is portable */
abc(&x[0]); /* this is portable */
```

If you need the address of a pointer to the array, declare a pointer to the array and then use the unary operator on the pointer, as shown in the following example:

```
char x[5], *y;

y = x;
abc(&y);
```

Null Pointers

The pointer value of NULL refers to a pointer value of 0 on all systems on which the SAS System currently runs. Therefore, comparisons of NULL to pointers set to 0 are portable.

Using SAS/TOOLKIT Tools to Avoid Problems

This section describes recommended coding techniques to avoid host dependencies in the following areas:

- zeroing memory
- maximum and minimum values
- floating-point magnitude and precision
- declaration of routines
- collating sequences
- checking missing values
- structure assignment.

Zeroing Memory

Never assume that storage areas are automatically set to zeros. When you allocate memory, use the SAS_XMEMZER routine to allocate and zero it at the same time. You can also use SAS_XMEMEX or SAS_XMALLOC with the XM_ZERO flag to zero memory while allocating it. If you have a block of memory that needs to be set to 0, use the SAS_ZZEROI routine.

Host-Specific Values

SAS/TOOLKIT software defines a number of MAC*xxxx* values to specify host-specific values. Chapter 24, "Miscellaneous Routines and Values," lists the macro values provided with SAS/TOOLKIT software to enable you to write your program without specifying host-specific values.

Floating-Point Magnitude and Precision

Floating-point numbers vary in their precision and magnitude, depending on the host system. For example, CMS and MVS hosts can store a maximum of approximately 1E75, while a PC/DOS host can store a maximum value of approximately 1E308. CMS and MVS hosts, however, provide 4 more bits of precision than PC/DOS hosts. Be careful to check for overflows in numeric algorithms. Use the MACBIG and MACSMALL values defined for each host to test for maximum and minimum values that can be stored on a host.

For more information on floating-point magnitude and precision, refer to "Details of Numeric Precision," (pp. 86-95) in Chapter 3, "Components of the SAS Language," in *SAS Language: Reference, Version 6, First Edition.*

Declaration of Routines

It is good practice to declare every routine used in a program even if the routine is internal to the source. It is especially important to declare the functions that return a value other than `int`, the default return attribute for functions. If your function returns a different type of value, not declaring the function can cause severe problems.

Use the U_PARMS macro and make the function declaration a prototype defining all arguments, as well as the return attribute, as shown in the following example:

```
int abc U_PARMS((int, double, char *));
```

On systems that have not implemented prototyping, this declaration is resolved to the following:

```
int abc();
```

But on systems where prototyping is implemented, the argument attributes are checked against the arguments passed when the function is used. When a function is called and one of the arguments has an incorrect data type, the compiler flags the incompatible data type so it can be corrected at the compilation stage.

Note: The U_PARMS macro is defined in the UWHOST member of the macro library provided with SAS/TOOLKIT software.

Collating Sequences

The SAS System runs on hosts that support at least two different collating sequences. CMS and MVS hosts use the EBCDIC sequence while most of the other hosts use standard ASCII. Therefore, you should not use code that assumes any specific collating sequence. For example, the following test for alphabetic characters works on only ASCII-based systems:

```
if ('A' <= x && x <= 'Z')
```

On EBCDIC systems, the alphabetic characters are not contiguous as they are on ASCII systems. Instead of performing this type of test, use the `isalpha` function.

If you have code that must be executed for a specific collating sequence, use the U_CHRSET symbol to test which collating sequence is in effect. The U_CHRSET macro is defined in the UWHOST member of the macro library provided with SAS/TOOLKIT software. The `#if` statement is evaluated at the time the program is compiled, so only the portion of the code that applies to the operating system compiling the program is actually compiled. The following example illustrates how to use the U_CHRSET macro:

```
#if (U_CHRSET == U_EBCDIC)
   statements for EBCDIC systems
#endif
#if (U_CHRSET == U_ASCII)
   statements for ASCII systems
#endif
```

Checking SAS Missing Values

Never use SAS missing values in arithmetic operations or even in comparisons. On some systems, any comparison to missing values is always true. In addition, some systems treat missing values as a very small number (not zero), so operations using missing values may cause an overflow or underflow.

If a variable in a SAS data set can possibly be a missing value, be sure to test for a missing value before performing any other operation with the value. The following examples on the left illustrate three statements that do not test for missing values that should. Use the SAS_ZMISS routine to test for missing values. This routine tests for the appropriate value on the host machine running the program. The examples on the right illustrate how to ensure that a value is not missing before performing any other tests or operations with the variable.

Incorrect Method	**Correct Method**
`if (x == 0)`	`if (!SAS_ZMISS(x) && x == 0)`
`if (x == y)`	`if (!SAS_ZMISS(x) &&` `   !SAS_ZMISS(y) && x == y)`
`x *= 5;`	`if (!SAS_ZMISS(x))` `   x *= 5;`

In the preceding examples that test multiple conditions, be sure to perform all the tests for missing values before comparing the variable to another value. In the C language, tests are performed from left to right and the analysis ends when the first untrue condition is encountered. Therefore, you must place the test for missing values before the other tests.

Note: The UWHOST member of the SAS/TOOLKIT macro library contains a macro for the MACMISSING value on your host. This value represents a standard missing value on your host, but it is not safe to compare the missing value to MACMISSING. Instead, use SAS_ZMISS for tests and use MACMISSING only when assigning a missing value to a variable.

Incorrect Method	**Correct Method**
`if (x == MACMISSING)` `   y = MACMISSING;`	`if (!SAS_ZMISS(x))` `   y = MACMISSING;`

Structure Assignment

Not all compilers allow structure assignment. You can copy the contents of one structure to another by using the `memcpy` C function or by using the SAS_ZMOVEI routine.

Scope

This section describes recommended coding techniques to avoid operating system dependencies in the following areas:

- use of `extern` variables
- structure scope.

Use of `extern` Variables

An item defined before the first function in a source module, but without the `extern` or `static` attribute, is an `extern`. The definition for this item should occur in only one source file. Any number of other modules can refer to the item as `extern`.

Structure Scope

Some compilers do not permit you to refer to a structure tag before the structure is defined. For example, the example on the left is not permitted on all compilers; the one on the right is portable:

Not Portable

```
   /* refers to XYZ before */
   /* it is declared       */
struct ABC {
   struct XYZ *a;
   long b;
   double c;
}

struct XYZ {
   int x;
   int y;
   int z;
}
```

Portable

```
   /* XYZ declared first   */
   /* and then used in ABC */
struct XYZ {
   int x;
   int y;
   int z;
}

struct ABC {
   struct XYZ *a;
   long b;
   double c;
}
```

If you need to include items at the beginning of your program that make references to structures that have not been defined yet, you can overcome the problem in the following ways:

- Store the structures referred to in the included code in separate files that can be included before the code that uses them.

- In the first structure, use a `char *` pointer instead of the actual structure pointer in the second structure, as shown in the following:

```
struct ABC {
   char *a;  /* use this instead of "struct XYZ *a" */
   long b;
   double c;
}
```

Include or define the second structure, XYZ, when appropriate. When you use the pointer in the first structure, cast `a`, the pointer-to-char, to a pointer to the second structure, as shown here.

```
struct XYZ {
   int x;
   int y;
   int z;
}
struct ABC example;
ptr    p;

example.a = (struct XYZ *)p;
```

Miscellaneous Techniques

This section describes recommended coding techniques to avoid host dependencies in the following areas:

- tab character
- general limitations
- continued lines in the source code
- using spacing with operators
- nesting comments.

Tab Character

Avoid using the tab character, `\t`, in character strings because it is not supported by all compilers. You can use the %t substitution character with character strings that are to be processed with the SAS_XP routines.

General Limitations

Different compilers have various limitations that you need to avoid in order to make your program completely portable. Table 25.1 summarizes limitations for all systems that support the SAS System. If you keep within these limitations, your program should work on all systems.

Table 25.1 *Summary of Limitations*

Capability	Limitation
`#include` nesting level	7
arguments to a macro	31
length of a literal	256
arguments to a function	31
total space for storing automatic variables	128K
array size	128K
size of source line	72♦

♦ The actual maximum is 80, but 72 is the recommended maximum to permit easier editing of the source file using ISPF under MVS.

Continued Lines in the Source Code

If your source line contains a literal that must be continued on the next line, use the backslash (\) as the last character of the line, then continue the literal in column 1 of the next line.

Note: If you transport the source file to a VMS system, you must ensure that a carriage-return or line-feed immediately follows the backslash. The C compiler under VMS issues error messages if the source file has trailing blanks after a backslash. Trailing blanks can occur in a file if you transport a fixed-length file to VMS without trimming the trailing blanks. The VMS system works primarily with variable-length files.

Using Spacing with Operators

Some compilers misinterpret the older-style method of coding operators. The older style permitted the assign-product operator, *=, to be written as =*. This operator could be confused with the assignment of a dereferenced pointer, as in the following examples:

Older Style	Possible Interpretations
`x=*y;`	`x *= y;` `x = *y;`

To avoid incorrect interpretations, use a space on either side of all operators or operator groups and avoid using the older style of operators such as =* and =+.

This problem can also occur when a macro is expanded by the compiler. For example, if you specify a macro and use it as shown on the left in the following example, the macro expansion resolves to the same indefinite form as shown on the right:

Use of Macro	Macro Expansion

`#define Y *z`
 	
`x=Y;`
	`x=*z;`

Once again, the form of the statement shown on the right can cause a misinterpretation. You can avoid this problem by using blank spaces on either side of the operator, as mentioned before. In addition, you can use parentheses in the definition of the macro to ensure clarity:

```
#define Y (*z)
```

This definition resolves to the following, which cannot be misinterpreted:

```
x=(*z);
```

Nesting Comments

Some compilers do not support nested comments so do not nest comments using this style of comments: /* */. To comment out a section of code that may include comments in this style, use the `#if` and `#endif` statements. For example, replace the example on the left with the second method of commenting, as shown in the example on the right:

Not Portable

```
/*
   /* description of section */
for (i = 1; i < 10; i++) {
   statements
}
*/
```

Portable

```
#if 0
   /* description of section */
for (i = 1; i < 10; i++) {
   statements
}
#endif
```

The `#if 0` statement ensures that all statements up to the `#endif` statement are not compiled.

Statement Labels

Ensure that all statement labels have a statement associated with them, even if it's a null statement. Unattached statement labels are not portable. For example, avoid unattached labels, as shown in the example on the left, and use a null statement as shown on the right:

Not Portable

```
labely:    /* unattached label*/
   { statements }
```

Portable

```
labelz: ;  /* null statement */
   { statements }
```

References

Kernighan, B.W. and Ritchie, D.M. (1978), *The C Programming Language*, Englewood Cliffs, NJ: Prentice-Hall, Inc.

Chapter 26 Writing Programs in the FORTRAN Language

Introduction

Many software applications must be written so the same program can be compiled and linked under more than one operating system. Developing programs so they will compile and run successfully under many systems, even if your current environment does not require portability, prevents the need to recode programs if your environment changes. SAS/TOOLKIT software supports the VS FORTRAN Version 2 compiler under MVS and CMS and the native FORTRAN compiler under VMS.

This chapter briefly discusses the facility included with SAS/TOOLKIT software that helps you to write portable programs in FORTRAN and also discusses a method for using pointers with FORTRAN.

FORTRAN Preprocessor

The FORTRAN language is implemented somewhat differently on the operating systems that support the SAS System. The primary differences are in the format and use of the INCLUDE statement and in the declaration of functions. To overcome these differences, SAS/TOOLKIT software includes a preprocessor program that overcomes operating system differences. This preprocessor is described in more detail in Chapter 7, "Building a Template for SAS/TOOLKIT Procedures in the FORTRAN Language."

Using Pointers with FORTRAN

The language used to develop most of SAS/TOOLKIT software and the SAS System is C. The C language is a call-by-value language, not a call-by-address language like FORTRAN. In a call-by-value language like C, you must pass any argument that needs to be updated by explicitly providing the address of the argument. This means that when you call a SAS/TOOLKIT routine and that routine needs to update a value you pass to it, you must explicitly provide a pointer to a data value, not the data value itself.

Unlike C, the FORTRAN language has no native pointer type. SAS/TOOLKIT software provides several routines to enable you to use pointers with FORTRAN. The following sections discuss the tools provided with SAS/TOOLKIT software to enable you to work with pointers in FORTRAN.

Obtaining the Address of a Variable

The IADDR function provided with SAS/TOOLKIT software returns the address of a numeric variable and stores the address in an INTEGER*4 variable.* The CADDR function returns the address of a character variable and provides the appropriate descriptive information about the variable. Both the CADDR and the IADDR functions are described in more detail in Chapter 24, "Miscellaneous Routines and Values."

When you use the IADDR and CADDR functions to obtain the address of a declared variable, you can easily access the value stored in the variable. If, however, you need to dynamically allocate storage and then work with the address of that storage area, FORTRAN presents more problems. Accessing the information pointed to by an address can be difficult. SAS/TOOLKIT software provides the LOCPTR routine to enable you to access dynamically allocated storage in FORTRAN more easily. The following section discusses this routine.

Using Dynamically Allocated Memory

Many FORTRAN programmers choose to use fixed-memory allocations because accessing dynamically allocated memory is difficult in FORTRAN. When you are using FORTRAN with SAS/TOOLKIT software, you have several tools to enable you to dynamically allocate and then access memory. First, use the SAS_XM routine to allocate memory when it is needed. To access the memory you allocate, use the LOCPTR routine, which treats the allocated memory as an indexed array.

Example of Using the LOCPTR Routine

The following example illustrates how to use the LOCPTR routine to access dynamically allocated memory. The LOCPTR routine requires a BASE array, the size of an array element, and the pointer to the allocated memory. In this example, the allocated memory is used to store floating-point values, so the BASE array is declared as REAL*8. The BASE array has only one element, but that is not a requirement; you can use any size array.

Note: You must define the dimensions of the BASE array to start at 0 for the LOCPTR routine to work correctly.

```
REAL*8 BASE(0:1)
INTEGER*4 P1,I1

P1 = SAS_XMEMGET(1000)
I1 = LOCPTR(BASE,8,P1)
```

* The FORTRAN compiler under VMS includes an IADDR function, but other compilers may not. The IADDR function is included with SAS/TOOLKIT software to ensure that this functionality is available on all systems.

The value returned by LOCPTR, I1, can be used as the index to the BASE array to access the first element in the array. So, the first element in the array is BASE(I1) and the *n*th element is BASE(I1+*n*−1).

Note: FORTRAN compilers do not issue warning or error messages when you access elements beyond the range of the specified dimension.

If the contents of the allocated memory is not floating-point values, simply change the declaration of the BASE array to the appropriate type.

When you pass the array element to a routine that uses it, you can simplify the index handling within the subroutine as shown here:

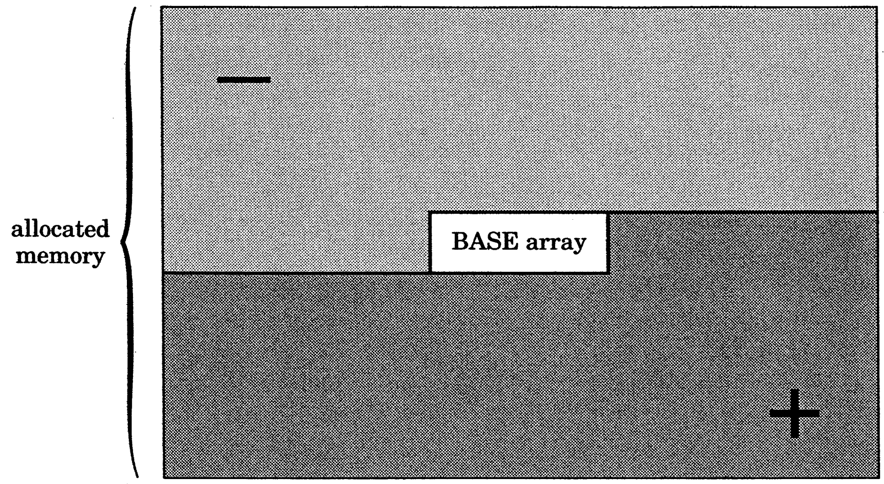
```
INTEGER*4 IBASE(0:1)
INTEGER*2 SBASE(0:1)
INTEGER*4 P2,P3,I2,I3

P2 = SAS_XMEMGET(2000)
I2 = LOCPTR(IBASE,4,P2)

P3 = SAS_XMEMGET(4000)
I3 = LOCPTR(SBASE,2,P3)

CALL DOIT(IBASE(I2),SBASE(I3))

SUBROUTINE DOIT(Y,Z)
INTEGER*4 Y(1)
INTEGER*2 Z(1)
```

In this example, the two array elements are passed to the subroutine, which treats the first element in your allocated memory as the first element in the array used by the subroutine. Therefore, to access the *n*th element of the allocated memory from within the subroutine, you can simply use Y(*n*) instead of having to offset from IBASE or SBASE.

How the LOCPTR Routine Works

LOCPTR uses the base array passed to it to calculate what element in the array corresponds to the dynamically allocated memory. Figure 26.1 shows that the index to the BASE array can be either positive or negative, depending on where the dynamically allocated memory is in relation to BASE.

Figure 26.1
Positive and Negative Indexes to Base Array



Consider this example. BASE is located at address 5000 (decimal), and you allocate 1000 bytes of memory with SAS_XMEMGET, which is stored at location 1000 (also decimal). P1, the address returned by SAS_XMEMGET, is 1000 or 4000 bytes before BASE. Each BASE element is 8 bytes long so there are 4000/8, or 500, elements between the allocated memory and BASE.* Since the memory comes before BASE, the first element is BASE(−500). This means that LOCPTR returns −500. If you pass BASE(−500) as an argument to a subroutine, the address of BASE(−500) is what is actually passed. When you treat the argument in the subroutine as an array, you can access array elements starting with an index of 1. In the example in the previous section, X(1) is actually at the same address as RBASE(−500), where the memory was allocated.

Note: FORTRAN compilers do not issue warning or error messages when you access elements beyond the range of the specified dimension.

* When you use one of the SAS_XM routines to allocate memory, it aligns the allocation on an 8-byte boundary, so any type of numeric array is sure of being aligned.

Chapter 27 Writing Programs in the PL/I Language

Introduction

Many software applications must be written so that the same program can be compiled and linked under more than one operating system. Developing programs that will compile and run successfully under many systems, even if your current environment does not require portability, prevents the need to recode programs if your environment changes.

SAS/TOOLKIT software supports the PL/I Optimizing Compiler under MVS and CMS and the native PL/I compiler under VMS. The PL/I compiler is an implementation of the ANSI PL/I Subset G standard, with some additions. One of these additions to the VMS compiler is a preprocessor, which is used very heavily by SAS/TOOLKIT software.

This chapter describes some of the requirements for writing programs in PL/I so the same source code can be used on all supported operating systems.

Tools Provided by SAS Institute

To assist you in writing more portable programs, SAS Institute provides a file that contains machine-dependent and operating-system-dependent information and some tools for handling this type of information.

- Use the portable preprocessor symbols defined in the MACPORT6 file. In particular, declare variables using the preprocessor variables listed in Table 27.1.
- When your program requires machine-dependent information, call the SAS_ZSYINF routine to complete the fields of the WZ_ENV structure. This structure contains information about the host system that enables you to avoid hard-coding values in your program.

Table 27.1 *Conversion Precisions and Sizes*

Preprocessor Symbol	Precision and Size
FXL	FIXED BIN(31)
FXS	FIXED BIN(15)
FTL	FLOAT BIN(53)
FTS	FLOAT BIN(21)
SIZE_FXL	4
SIZE_FXS	2
SIZE_FTL	8
SIZE_FTS	4
SIZE_PTR	4

Limitations for Portable Programs

To ensure that a source program is portable to all systems that support SAS/TOOLKIT software, you need to adhere to the restrictions of subset G of the PL/I language.

Note: Subset G restrictions are not required for the PL/I Optimizing Compiler for CMS and MVS, but if you do not adhere to the restrictions described here, you cannot transport your program to a system that does have subset G restrictions.

The following list summarizes some of the more common restrictions of PL/I subset G coding limitations. Refer to the reference manual for PL/I subset G for your operating system for more detailed information on these and other restrictions.

- Do not use the LIKE attribute.
- All subroutine arguments must be declared, and they must be declared exactly the same way in both the subroutine and the ENTRY declaration in the calling routine. This point is discussed in more detail in "Declare Subroutines" later in this chapter.
- Do not use the #, $, or @ symbols in names.
- Do not use division of two fixed-point binary values except by the DIVIDE function, as shown here:

```
Q=DIVIDE(NUM,DEN,31,0);
```

- □ Function calls must always have parentheses, even if they have no arguments. Here is an example:

  ```
  PTR = SAS_NUL();
  ```

  Note that this restriction doesn't apply to subroutines, which do not return a value. Subroutines do not require a null argument list.

- □ Do not use replicated string constants such as

  ```
  STRING=(5)'-';
  ```

- □ DO statements cannot contain comma lists or UNTIL clauses such as

  ```
  DO I=1,2 UNTIL(X=0);
  ```

- □ DO statements with an index variable must use integer (FXL or FXS) variables or values only.
- □ Do not use programmer-named conditions or the FINISH condition.
- □ An assignment statement can have only one target. This double assignment method is not permitted:

  ```
  X,Y = 0;  /* not portable */
  ```

- □ If you use the INITIAL attribute to set initial values, you must use the STATIC attribute as well:

  ```
  DCL X(3) FXL STATIC INIT(1,2,3); /* permitted */
  ```

▶ ***Caution*** .... ***The value of a STATIC variable should not be changed.***
Changing a STATIC variable makes your program nonreentrant. ▲

- □ MAX and MIN must have exactly two arguments. Do not use this format for the MAX statement:

  ```
  X = MAX(A,B,C);    /* not portable */
  ```

  Use this format instead:

  ```
  X = MAX(MAX(A,B),C);  /* portable */
  ```

- □ Avoid using the UNSPEC function because the layout of bits for a variable may not be consistent between operating systems. This is especially true for EBCDIC and ASCII systems, or for integer representations under MVS and CMS versus VMS.
- □ Bit variables are permitted, but don't attempt to overlay them with character variables and compare them to other character variables.
- □ Do not use COMPLEX data types.
- □ Do not use CONTROLLED storage.
- □ Do not use area or offset variables.
- □ Do not use REFER.

- Do not use DEFINED with POSITION or ISUB.
- Do not use subtasking.
- Do not use optimizer extensions such as SELECT CASE.
- Do not use multiple labels or subscripted labels in PROC statements.
- END statements cannot do multiple closures. They can have a label.
- Do not use the DEFAULT statement.
- THEN and ELSE cannot be labeled.

Overcoming Host Differences

This section discusses some differences in PL/I on CMS and MVS versus VMS. The following sections describe the requirements of each system and recommend coding techniques or facilities of SAS/TOOLKIT software that can help you overcome the differences.

Requirements for the OPTIONS Statement

When you write a PL/I procedure under MVS or CMS, you need to have the following OPTIONS statement for the main module:

```
OPTIONS(MAIN,REENTRANT) REORDER
```

You need this OPTIONS statement on other modules:

```
OPTIONS(REENTRANT) REORDER
```

However, the VMS compiler does not recognize these options, so in order to develop portable source code, use the following preprocessor variables:

```
PORTMAIN
PORTPROC
```

These preprocessor variables are declared in the MACPORT6 member of the macro library provided with SAS/TOOLKIT software.

Under MVS and CMS, these variables expand to the two OPTIONS statements illustrated earlier under VMS, these variables are declared as null statements, so they do not affect the compilation.

Note: When compiling PL/I modules under CMS and MVS, be sure to use the OPT option when you invoke the compiler.

LABEL Arrays

If you have a LABEL array that you use with a branch table, you must declare the array under MVS and CMS, but you must omit the declaration under VMS:

CMS and MVS Format	**VMS Format**
`DCL XXX(3) LABEL;`	`GOTO XXX(I);`
`GOTO XXX(I);`	

To write this portion of the program portably, use the MACHINE preprocessor variable as follows:

```
%IF MACHINE='IBM' %THEN DO;
DCL XXX(3) LABEL;
%END;

GOTO XXX(I);
```

The MACHINE preprocessor variable is declared in the MACPORT6 member of the macro library provided with SAS/TOOLKIT software. This variable is set to IBM for MVS and CMS environments and to VAX for the VMS environment.

Declare All Variables

Be sure that you declare all variables because the default attribute for the undeclared variables is different in different environments. For example, under MVS and CMS, the compiler assigns FXS to any variable with a name that begins with the letters I through N and FLOAT DEC(6) to any other undeclared variable whose type cannot be determined from the context of its use. The VMS compiler assumes FXL for any undeclared variable.

For example, be sure to declare both the array and the pointer in this type of declaration:

```
DCL XPTR PTR;
DCL X CHAR(10) BASED(XPTR);
```

If you omit the pointer declaration under VMS, the compiler assumes that XPTR is FXL and the compilation fails. The IBM compiler properly determines from context that XPTR is a POINTER.

Declare Subroutines

Be sure to declare all subroutines used in your program. The CMS and MVS compiler only warns you when a subroutine is not declared, but under VMS the compilation fails.

Declaring subroutines portably presents some difficulties if the subroutine has a varying number of arguments. Under VMS, you must declare each argument passed to a routine, but you can use the ANY description for an argument to allow any type of attribute. Under MVS and CMS, the ANY description is not recognized, but you do not have to declare the arguments for a routine. The following examples illustrate the different forms permitted under these environments:

MVS and CMS Format	**VMS Format**
`DCL ABC ENTRY;`	`DCL ABC ENTRY(ANY,ANY,ANY);`
`CALL ABC(1,2,3);`	`CALL ABC(1,A,2);`

To overcome this difference, you can use the MACHINE preprocessor variable to test the host environment and declare the routine appropriately for each host, as shown in the following example:

```
%INCLUDE MACPORT6;
%IF MACHINE='IBM' %THEN %DO;
DCL ABC ENTRY;
%END;
%ELSE %DO;
DCL ABC ENTRY(ANY,ANY,ANY);
%END;
```

The MACHINE preprocessor variable is declared in the MACPORT6 member of the macro library provided with SAS/TOOLKIT software. This variable is set to IBM for MVS and CMS environments and to VAX for the VMS environment.

Passing Constants to Routines

When you pass a constant to a subroutine, the default attribute for the constant in the subroutine that receives it is different under different operating systems. For example, under VMS the default attribute is FXL, and under MVS and CMS, the default attribute is FIXED DEC (that is, packed decimal). To ensure that the value passed to a subroutine has the correct attributes, you can do either of the following:

- □ Use the FIXED, FLOAT, or DESCRIPTOR function to declare the correct attributes for the constant, as illustrated here:

```
CALL ABC(FIXED(1,31),FLOAT(2.4,53),DESCRIPTOR('DEF'));
```

- □ Assign the value to a previously declared variable and pass the variable to the subroutine, as shown in the following example:

```
DCL X FXL;
DCL Y FTL;
DCL Z CHAR(3);

X = 1;
Y = 2.4;
Z = 'DEF';
CALL ABC(X,Y,Z);
```

PL/I Descriptors for Character Variables

PL/I compilers use a character descriptor, commonly known as the *dope vector,* to provide information about a character variable. This dope vector contains a pointer to the character data and the length of the data. The layout of this dope vector and how it is used differs in different environments.

Under CMS and MVS, when you declare an entry for a subroutine that processes character variables, such as the one illustrated in the following example, both character variables are handled in the same way:

```
DCL ABC ENTRY(CHAR(8),CHAR(*));
```

That is, for both character variables, the ABC routine receives a pointer to the dope vector for each variable. Under VMS, however, the ABC routine receives a pointer to the dope vector for the second variable, but for the first variable, it receives a pointer to the actual data. Since the length of the first variable is known (8), VMS references only the string itself, not a dope vector.

Subroutines versus Functions

Under CMS and MVS, if you declare a function, you can use the function with or without the return value, as shown here:

```
DCL ABC ENTRY(FIXED BIN(31)) RETURNS(FIXED BIN(31));

X=ABC(Y);  /* correct use of function */
CALL ABC(Y);    /* valid only for CMS and MVS */
```

Under VMS, if you declare a function, it must be used with the return value. If you attempt to use the second form shown above, the VMS compiler issues error messages.

To be sure that your program is portable, use only the first form of the function call shown in the example.

Determining the Size of Elements

Under CMS and MVS, the CSTG function returns the size of an element. Under VMS, the native PL/I SIZE function provides the size, but SAS/TOOLKIT software includes a macro that defines CSTG to equate to SIZE. Therefore, you can use the following statements on all operating systems to portably determine the size of the ABC structure:

```
DCL 1 ABC,
      2 X FIXED BIN(31),
      2 Y FTL;

Z = CSTG(ABC);
```

In addition, you can use the macro variables listed in Table 27.1 for the size of certain data types.

Obtaining a NULL Value

The NULL built-in function does not return a true zero pointer on all systems. Therefore, you use the SAS_NUL function provided with SAS/TOOLKIT software to get a true NULL value. To simplify the use of this function, declare a local pointer variable and set its value to NULL. Then simply assign its value to any variable that needs to be set to NULL.

```
DCL NUL0 PTR;
NUL0 = SAS_NUL();

X = NUL0;
Y = NUL0;
```

Other Recommended Practices

The following sections suggest coding techniques that make your program more efficient or more robust.

Overlaying Descriptions

To overlay a description on an area of memory, use the BASED(ADDR(*variable*)) method, as shown here, instead of the DEFINED attribute.

```
DCL X FIXED BIN(31), XB CHAR(4) BASED(ADDR(X));
```

Be careful when overlaying descriptions, however. For example, you cannot always expect XB(4) to contain the low-order 8 bits of the FIXED BIN number in X. This will be the case under MVS and CMS; but under VMS, the number is stored differently.

Arrays as Arguments

Avoid passing arrays as arguments. Instead, pass a pointer to the array since this works more efficiently. If you must pass an array instead of a pointer, declare the array in the subroutine with (*).

In addition, use 0-based arrays whenever possible, as illustrated here, since 0-based arrays generate much more efficient code:

Less Efficient

```
DCL X(4) FIXED BIN(31);

DO I=1 TO 4;
   X(I) = X(I) * 2;
   END;
```

More Efficient

```
DCL X(0:3) FIXED BIN(31);

DO I=0 TO 3;
   X(I) = X(I) * 2;
   END;
```

Pointer Arithmetic

Pointer arithmetic is not actually supported as a PL/I process although the process shown on the left in the following example has been used to simulate pointer arithmetic. Note that this process is not recommended because integer arithmetic doesn't necessarily match pointer arithmetic. Instead, use the alternate method shown on the right to avoid problems.

Avoid Using

```
DCL ABC CHAR(32);
DCL X FIXED BIN(31),
    XPTR PTR BASED(ADDR(X));
XPTR = ADDR(ABC);
X = X + 16;
```

Recommended Method

```
DCL ABC CHAR(32);
DCL XXX CHAR(0:1000) BASED;
DCL XPTR PTR;

XPTR = ADDR(ABC);
XPTR = ADDR(XPTR->XXX(16));
```

The recommended method returns the address of the element that has the number by which you want to increment the pointer.

Miscellaneous Recommendations

In addition to the specific changes listed in the preceding sections, you can employ the following programming techniques to make your programs more portable:

- Use ALIGNED for bit strings where possible.
- Be careful not to make EBCDIC or ASCII assumptions. For example, if you are verifying that a character is a numeric character, use a test that checks both ends of the range. The example on the left in the following example works in most cases on EBCDIC systems, but is not a valid test on ASCII systems. The example on the right is valid for both EBCDIC and ASCII systems:

  Not Portable

  ```
  IF X>='0' THEN DO;
  ```

  Portable

  ```
  IF X>='0' & X<='9' THEN DO;
  ```

- Do not assume that a sorted list of STATIC INIT values is sorted when it is transported to another operating system. If it is sorted in EBCDIC order, it may or may not be sorted in ASCII order. This assumption can cause serious problems with binary searches.
- Remember that programs developed under CMS or MVS may have sequence numbers in columns 73 through 80 or carriage-control characters in column 1. These characters must be stripped off when the program is transported to VMS. If you simply replace these columns with blanks, you may introduce unwanted blanks into your program; this is especially a problem when a character string is continued from one line to another. To avoid this problem, you can create your transported file by reading only columns 2 through 72 of the original file and then adding appropriate ASCII characters for carriage control.

In addition, if you develop a program under the VMS operating system, it is a good idea to use only columns 2 through 72 to make your program portable to the CMS or MVS environment.

Part 8

Operating System Appendices

Appendix 1 Using SAS/TOOLKIT™ Software under CMS

Introduction

This appendix provides sample EXECs for compiling and link-editing your procedure, function, CALL routine, informat, or function in the various supported languages. Under CMS, you can use the following compilers:

- the SAS/C Compiler
- the PL/I Optimizing Compiler
- the VS FORTRAN Version 2 Compiler
- the IBM 370 Version 2 H-level Assembler.

In addition, this appendix discusses how to use user-written SAS modules after they are compiled and how to make those modules available to other sites.

Accessing User-Written SAS Modules

This section discusses how to access user-written modules and how to make these modules available to other sites.

Specifying Where to Find Modules

After you have compiled and link-edited your program, you can access it by using the SASLOAD= SAS system option to specify the load library containing your program. When you specify a SASLOAD library, the SAS System searches this library for a load module before it searches the standard SAS load libraries.

You must include the SASLOAD= option in one of the following locations:

- your configuration file (usually CONFIG SAS)
- the system configuration file (SASPROC SAS)
- in the SAS command as you invoke the system.

Note that you cannot use an OPTIONS statement or window within a SAS session to invoke the option.

In addition to identifying your own load library, you must be sure that the necessary load libraries for SAS/TOOLKIT software are also specified in a SASLOAD= option. The SASLOAD= options for these libraries usually are included in the SASPROC SAS file, but they are illustrated here in case they have not been added to your site's SASPROC SAS file.

The following example illustrates the options needed to include the SAS/TOOLKIT load libraries and your own load library (in this example SASABC LOADLIB) in the search path for the SAS System.

```
SASLOAD='SASPROCI LOADLIB *'
SASLOAD='SASUSERP LOADLIB *'
SASLOAD='SASABC LOADLIB *'
```

If your user-written module needs to be available to other users at your site, ask your SAS Software Representative to include the SASLOAD= option for your load library in the SASPROC SAS file.

Distributing Modules to Other SAS Sites

If you distribute a module you have written to other SAS sites, you may also need to make available the SASPROCI module, supplied with SAS/TOOLKIT software. You are free to redistribute SASPROCI, since it is treated as a transient module. If the site to which you are distributing your SAS module has Release 6.07 or later of the SAS System, you probably do not need to supply the SASPROCI module. If you do not supply SASPROCI and the site receiving your module does not already have it, your procedure or IFFC will not run at that site.

Note: You are *not* permitted to redistribute any other components of SAS/TOOLKIT software.

Overview of Compiling and Linking

The process for compiling and linking a user-written SAS module is very similar for all types of modules and any language. This section describes the steps in general. The remainder of this appendix provides the details for compiling and linking each type of SAS module that you can write in each language.

This section also describes the LKEDPV6 EXEC, which provides a convenient method for compiling and linking.

Steps for Compiling and Linking

The general steps for compiling and linking are as follows:

1. Create one or more TEXT files with the statements needed by the linkage editor. These TEXT files are illustrated for each language and each type of module in later sections of this appendix.
2. Modify one of the sample EXECs provided with SAS/TOOLKIT software with the appropriate file information. Each EXEC compiles and links a specific type of module for a single language. Table A1.1 lists these EXECs. Every EXEC performs the following tasks:
   - □ builds a TEXT file with the SAS statements to run the USERPROC procedure. This step creates the program constants object module needed to link the module.
   - □ runs the SAS statements created in the previous step.
   - □ issues the GLOBAL command for the necessary files.
   - □ compiles each source module.
   - □ issues FILEDEFs required by the LKED command.
   - □ creates the executable module by executing the LKEDPV6 EXEC. This EXEC is described in more detail later in this section.
3. Run the EXEC you modified in step 2.

   The EXECs provided with SAS/TOOLKIT software are listed in Table A1.1.

Table A1.1 *EXECs for Compiling and Linking*

EXEC Name	Language	Module Type
IFFJOBFA	assembler	format
IFFJOBFC	SAS/C	format
IFFJOBFF	FORTRAN	format
IFFJOBFP	PL/I	format
IFFJOBIA	assembler	informat
IFFJOBIC	SAS/C	informat
IFFJOBIF	FORTRAN	informat
IFFJOBIP	PL/I	informat
IFFJOBUA	assembler	function or CALL routine
IFFJOBUC	SAS/C	function or CALL routine
IFFJOBUF	FORTRAN	function or CALL routine
IFFJOBUP	PL/I	function or CALL routine

(*continued*)

Table A1.1 (continued)

EXEC Name	Language	Module Type
PRCJOBA2	assembler	procedure
PRCJOBC2	SAS/C	procedure
PRCJOBF2	FORTRAN	procedure
PRCJOBP2	PLI	procedure

Each of the EXECs listed in Table A1.1 executes another EXEC, the LKEDPV6 EXEC, which is described in more detail in the next section.

▶ ***Caution*** .... ***Do not use a compiler option that globally causes data elements to be unaligned.*** SAS/TOOLKIT software expects the structures it uses to be aligned. If you want a structure used only by your program to be unaligned, you can use a compiler option that affects only that structure (if the compiler you are using provides such an option). For example, the __NOALIGNMEM option for the SAS/C Compiler can be used with a structure as follows:

```
__NOALIGNMEM struct ABC {
   structure declaration
   }
```

Consult the documentation for the compiler you are using to determine if an option similar to this one is available. ▲

Using the LKEDPV6 EXEC

SAS/TOOLKIT software includes a useful EXEC called LKEDPV6 to simplify the link-editing of procedures and IFFCs. This EXEC is customized for your site at the time that SAS/TOOLKIT software is installed. Ask your SAS Software Representative to help you locate this sample EXEC.

The syntax for invoking this EXEC is the following:

LKEDPV6 *textfile module language* <PROC> <IFF>

textfile
: is the filename of a TEXT file containing statements for the linkage editor. These TEXT files are described in detail for each type of module and each language in later sections in this appendix.

module
: is the name of the LOADLIB member to be created. This name should match the filename of the program constants TEXT file. The program constants TEXT file is specified in the MODNAME= argument of the PROC USERPROC statement.

language
: must be SASC (or just C), PLI, FORTRAN, or ASM.

PROC
: must be specified if you are linking a PL/I or FORTRAN procedure. This flag indicates that a special section of the EXEC must be executed to ensure that all external references are correctly resolved by the CMS LKED command.

IFF
: must be specified if you are linking a PL/I or FORTRAN IFFC. This flag indicates that a special section of the EXEC must be executed to ensure that all external references are correctly resolved by the CMS LKED command.

Compiling and Link-editing C Programs

This section illustrates sample REXX EXECs provided with SAS/TOOLKIT software to compile and link-edit procedures and IFFCs with the SAS/C Compiler and lists the contents of the TEXT files needed for input to the linking steps. In addition, this section includes a discussion of CMS versions and shared segments that may affect your programs.

Note: The SAS/C Compiler requires a special step, called the CLINK step, before you execute the linkage editor. The CLINK step helps to ensure that all external references are correctly resolved by the linkage editor. When you are compiling a SAS module with the SAS/C Compiler, you need to create an extra TEXT file with input to the CLINK step. This TEXT file is in addition to the TEXT file required by the linkage editor. All of the sections describing C programs illustrate both of these TEXT files.

All of the EXECs used to compile and link SAS/C programs use the following file names:

LC370STD TXTLIB and LC370BAS TXTLIB
: are the SAS/C resident libraries.

SASCMAC MACLIB
: is the SAS/C macro library.

TOOLKIT LOADLIB
: is the executable library for SAS/TOOLKIT software.

Your site may have these libraries installed under different names. In that case, you need to modify the EXECs provided with SAS/TOOLKIT software to use the correct names at your site for these libraries.

SAS/C Procedures

This section describes the two TEXT files needed to compile and link C procedures with the SAS/C Compiler and illustrates the PRCJOBC2 EXEC for compiling and linking.

Required TEXT Files

Create a TEXT file to specify input to the CLINK step. A TEXT file must contain the following statements:

```
INCLUDE text1
...
INCLUDE textn
INCLUDE SYSLIB(PRCINT)
```

where *text1* through *textn* are all of the compiled modules of your procedure; that is, they are the TEXT files created by the SAS/C Compiler. For procedures, you need to specify both the compiled procedure code and the compiled grammar function. The PRCINT module is a required part of SAS/TOOLKIT software.

The second text file provides input to the linkage editor. The name of this file appears as the first operand when you execute the LKEDPV6 EXEC (see the last statement in the PRCJOBC2 EXEC listed in the next section). This file contains the following statements:

```
INCLUDE program-constants
INCLUDE CLINK370
ENTRY   ENTRY
NAME    module(R)
```

where *program-constants* is the name of the file created by the PROC USERPROC step. This name should match *module* in the NAME statement, and both names should be the same as the value of the MODNAME= argument in the PROC USERPROC statement. Note that this must be the first line in the TEXT file. The CLINK370 file is the file created by the CLINK step.

The PRCJOBC2 EXEC

This REXX EXEC illustrates how to compile and link-edit a procedure with the SAS/C Compiler. This sample EXEC uses the following names for files:

EXAMPLE	is the name of the executable file created by the linkage editor.
EXAMPLEC	is the name of the file containing the source code to be compiled.
EXMPLG	is the name your procedure uses to call the grammar function.
EXMPLGC	is the name of the grammar function created by the USERPROC procedure.
EXMPLNKC	is the name of the TEXT file containing the statements for the linkage editor.
PROCLINK	is the name of the TEXT file containing the statements for the CLINK step.

You can change any of these names, but be sure that all occurrences of the name are changed.

Listing A1.1
PRCJOBC2 EXEC

```
/* REXX EXEC to compile and linkedit the EXAMPLE procedure in SAS/C */

/*-----create SAS program to build program constants object-----*/
queue 'PROC USERPROC OBJECT=''EXAMPLE TEXT'' MODNAME=EXAMPLE'
queue '     MODTYPE=PROC;'

/*-----create SAS program to build grammar function in SAS/C----*/
queue 'PROC USERPROC GRAMMAR=''EXAMPLE GRM'' GRAMFUNC=EXMPLG'
queue '     FUNCFILE=''EXMPLGC C'' NOPRINT'
queue '     GRAMLANG=C INCLLIB=''GRMMAC MACLIB *''; RUN;'
queue

/*-----copy SAS program to TEMP SAS temporary file-----*/
'EXECIO * DISKW TEMP SAS A3 1 F 80 (FINIS'

/*-----invoke the SAS System to create these entities-----*/
'EXEC SAS6 TEMP (NOLOG)'
```

(continued on next page)

Listing A1.1 (continued)

```
/*-----necessary globals for SAS/C compilation-----*/
'GLOBAL TXTLIB LC370STD LC370BAS'
/*-----necessary global for compilation of SAS/C toolkit items-----*/
'GLOBAL MACLIB SASCMAC'

/*-----compile the procedure and grammar functions-----*/
'EXEC LC370 EXAMPLEC'
'EXEC LC370 EXMPLGC'

'FI SYSLIB DISK TOOLKIT LOADLIB *'
/*-----create CLINK370 TEXT file for subsequent linkedit-----*/
'CLINK PROCLINK'

/*-----request final linkedit to create EXAMPLE LOADLIB-----*/
'FILEDEF EXAMPLE DISK EXAMPLE TEXT * (PERM'
'EXEC LKEDPV6 EXMPLNKC EXAMPLE SASC'

exit
```

Note: When you use this EXEC, you may receive warnings about unresolved references after the CLINK step. These should be resolved by the linkage editor step.

SAS/C Functions and CALL Routines

This section describes the two TEXT files needed to compile and link C functions and CALL routines with the SAS/C Compiler and illustrates the IFFJOBUC EXEC for compiling and linking.

Required TEXT Files

Create a TEXT file to specify input to the CLINK step. The TEXT file must contain the following statements:

```
INCLUDE text
INCLUDE SYSLIB(IFFEXTC)
```

where *text* is the compiled function; that is, the TEXT file created by the SAS/C Compiler. The IFFEXTC module is needed only if you do not include an IFFEXT routine in your function source code.

The second text file provides input to the linkage editor. The name of this file appears as the first operand when you execute the LKEDPV6 EXEC (see the last statement in the IFFJOBUC EXEC listed in the next section). This file contains the following statements:

```
INCLUDE program-constants
INCLUDE CLINK370
INCLUDE SYSLIB(IFFINT)
INCLUDE SYSLIB(SASCFACE)
INCLUDE SYSLIB(PRCINT)
ENTRY   IFFSASC
NAME    module(R)
```

where *program-constants* is the name of the file created by the PROC USERPROC step. This name should match *module* in the NAME statement, and both names should be the same as the value of the MODNAME= argument in the PROC USERPROC statement. Note that this must be the first line in the TEXT file. The CLINK370 file is the file created by the CLINK step. The IFFINT and SASCFACE modules are required parts of SAS/TOOLKIT software. You need to specify PRCINT only when your function uses SAS_X or SAS_Z routines.

The IFFJOBUC EXEC

This REXX EXEC illustrates how to compile and link-edit a function or CALL routine with the SAS/C Compiler. This sample EXEC uses the following names for files:

FUNCC
: is the name of the file containing the source code to be compiled.

FUNCLINK
: is the name of the TEXT file containing the statements for the CLINK step.

FUNCLNKC
: is the name of the TEXT file containing the statements for the linkage editor.

UWUTESTF
: is the name of the program constants object file and the executable file created by the linkage editor.

You can change any of these names, but be sure that all occurrences of the name are changed.

Listing A1.2
IFFJOBUC EXEC

```
 /* REXX EXEC to compile and linkedit the sample functions in SAS/C */

 /* ensure C compiler disk(s) are accessed */

 /*-----create SAS program to build program constants object-----*/
queue 'PROC USERPROC OBJECT=''UWUTESTF TEXT'' MODNAME=UWUTESTF'
queue '     MODTYPE=FUNCTION;'
queue

 /*-----copy SAS program to TEMP SAS temporary file-----*/
'EXECIO * DISKW TEMP SAS A3 1 F 80 (FINIS'

 /*-----invoke the SAS System to create these entities-----*/
'EXEC SAS6 TEMP (NOLOG)'

 /*-----necessary globals for SAS/C compilation-----*/
'GLOBAL TXTLIB LC370STD LC370BAS'
 /*-----necessary global for compilation of SAS/C toolkit items-----*/
'GLOBAL MACLIB SASCMAC'

 /*-----compile the function code-----*/
'EXEC LC370 FUNCC (-I'

'FILEDEF SYSLIB DISK TOOLKIT LOADLIB * (PERM RECFM U BLOCK 19096)'
 /*-----create CLINK370 TEXT file for subsequent linkedit-----*/
'CLINK FUNCLINK'

 /*-----request final linkedit to create UWUTESTF LOADLIB-----*/
'FILEDEF UWUTESTF DISK UWUTESTF TEXT * (PERM'
'EXEC LKEDPV6 FUNCLNKC UWUTESTF SASC'

exit
```

Note: When you use this EXEC, you may receive warnings about unresolved references after the CLINK step. These should be resolved by the linkage editor step.

SAS/C Informats

This section describes the two TEXT files needed to compile and link C informats with the SAS/C Compiler and illustrates the IFFJOBIC EXEC for compiling and linking.

Required TEXT Files

Create a TEXT file to specify input to the CLINK step. The TEXT file must contain the following statements:

```
INCLUDE text
INCLUDE SYSLIB(IFFEXTC)
```

where *text* is the compiled informat; that is, the TEXT file created by the SAS/C Compiler. The IFFEXTC module is needed only if you do not include an IFFEXT routine in your informat source code.

The second text file provides input to the linkage editor. The name of this file appears as the first operand when you execute the LKEDPV6 EXEC (see the last statement in the IFFJOBIC EXEC listed in the next section). This file contains the following statements:

```
INCLUDE program-constants
INCLUDE CLINK370
INCLUDE SYSLIB(IFFINT)
INCLUDE SYSLIB(SASCFACE)
INCLUDE SYSLIB(PRCINT)
ENTRY   IFFSASC
NAME    module(R)
```

where *program-constants* is the name of the file created by the PROC USERPROC step. This name should match *module* in the NAME statement, and both names should be the same as the value of the MODNAME= argument in the PROC USERPROC statement. Note that this must be the first line in the TEXT file. The CLINK370 file is the file created by the CLINK step. The IFFINT and SASCFACE modules are required parts of SAS/TOOLKIT software. You need to specify PRCINT only when your informat uses SAS_X or SAS_Z routines.

The IFFJOBIC EXEC

This REXX EXEC illustrates how to compile and link-edit an informat with the SAS/C Compiler. This sample EXEC uses the following names for files:

INFMLINK
: is the name of the TEXT file containing the statements for the CLINK step.

INFMLNKC
: is the name of the TEXT file containing the statements for the linkage editor.

INFMTC
: is the name of the file containing the source code to be compiled.

UWITESTI
: is the name of the program constants object file and the executable file created by the linkage editor.

You can change any of these names, but be sure that all occurrences of the name are changed.

Listing A1.3
IFFJOBIC EXEC

```
/* REXX EXEC TO COMPILE AND LINKEDIT THE SAMPLE INFORMATS IN SAS/C */

/*-----create SAS program to build program constants object-----*/
queue 'PROC USERPROC OBJECT=''UWITESTI TEXT'' MODNAME=UWITESTI'
queue '     MODTYPE=INFORMAT;'
queue

/*-----copy SAS program to TEMP SAS temporary file-----*/
'EXECIO * DISKW TEMP SAS A3 1 F 80 (FINIS'

/*-----invoke the SAS System to create these entities-----*/
'EXEC SAS6 TEMP (NOLOG)'

/*-----necessary globals for SAS/C compilation-----*/
'GLOBAL TXTLIB LC370STD LC370BAS'
/*-----necessary global for compilation of SAS/C toolkit items-----*/
'GLOBAL MACLIB SASCMAC'

/*-----compile the function code-----*/
'EXEC LC370 INFMTC (-I'

'FI SYSLIB DISK TOOLKIT LOADLIB *'
/*-----create CLINK370 TEXT file for subsequent linkedit-----*/
'CLINK INFMLINK'

/*-----request final linkedit to create UWITESTI LOADLIB-----*/
'FILEDEF UWITESTI DISK UWITESTI TEXT * (PERM'
'EXEC LKEDPV6 INFMLNKC UWITESTI SASC'

exit
```

Note: When you use this EXEC, you may receive warnings about unresolved references after the CLINK step. These should be resolved by the linkage editor step.

SAS/C Formats

This section describes the two TEXT files needed to compile and link C formats with the SAS/C Compiler and illustrates the IFFJOBFC EXEC for compiling and linking.

Required TEXT Files

Create a TEXT file to specify input to the CLINK step. The TEXT file must contain the following statements:

```
INCLUDE text
INCLUDE SYSLIB(IFFEXTC)
```

where *text* is the compiled format, that is, the TEXT file created by the SAS/C Compiler. The IFFEXTC module is needed only if you do not include an IFFEXT routine in your format source code.

The second text file provides input to the linkage editor. The name of this file appears as the first operand when you execute the LKEDPV6 EXEC (see the last statement in the IFFJOBFC EXEC listed in the next section). This file contains the following statements:

```
INCLUDE program-constants
INCLUDE CLINK370
```

```
INCLUDE SYSLIB(IFFINT)
INCLUDE SYSLIB(SASCFACE)
INCLUDE SYSLIB(PRCINT)
ENTRY   IFFSASC
NAME    module(R)
```

where *program-constants* is the name of the file created by the PROC USERPROC step. This name should match *module* in the NAME statement, and both names should be the same as the value of the MODNAME= argument in the PROC USERPROC statement. Note that this must be the first line in the TEXT file. The CLINK370 file is the file created by the CLINK step. The IFFINT and SASCFACE modules are required parts of SAS/TOOLKIT software. You need to specify PRCINT only when your format uses SAS_X or SAS_Z routines.

The IFFJOBFC EXEC

This REXX EXEC illustrates how to compile and link-edit a format with the SAS/C Compiler. This sample EXEC uses the following names for files:

FMTC — is the name of the file containing the source code to be compiled.

FMTLINK — is the name of the TEXT file containing the statements for the CLINK step.

FMTLNKC — is the name of the TEXT file containing the statements for the linkage editor.

UWFONEOF — is the name of the program constants object file and the executable file created by the linkage editor.

You can change any of these names, but be sure that all occurrences of the name are changed.

Listing A1.4
IFFJOBFC EXEC

```
 /* REXX EXEC TO COMPILE AND LINKEDIT THE SAMPLE   FORMATS IN SAS/C */

 /*-----create SAS program to build program constants object-----*/
queue 'PROC USERPROC OBJECT=''UWFONEOF TEXT'' MODNAME=UWFONEOF'
queue '     MODTYPE=FORMAT;'
queue

 /*-----copy SAS program to TEMP SAS temporary file-----*/
'EXECIO * DISKW TEMP SAS A3 1 F 80 (FINIS'

 /*-----invoke the SAS System to create these entities-----*/
'EXEC SAS6 TEMP (NOLOG)'

 /*-----necessary globals for SAS/C compilation-----*/
'GLOBAL TXTLIB LC370STD LC370BAS'
 /*-----necessary global for compilation of SAS/C toolkit items-----*/
'GLOBAL MACLIB SASCMAC'

'FI SYSLIB DISK TOOLKIT LOADLIB *'
 /*-----compile the function code-----*/
'EXEC LC370 FMTC (-I'

 /*-----create CLINK370 TEXT file for subsequent linkedit-----*/
'CLINK FMTLINK'

 /*-----request final linkedit to create UWFONEOF LOADLIB-----*/
'FILEDEF UWFONEOF DISK UWFONEOF TEXT * (PERM'
'EXEC LKEDPV6 FMTLNKC UWFONEOF SASC'

exit
```

Note: When you use this EXEC, you may receive warnings about unresolved references after the CLINK step. These should be resolved by the linkage editor step.

Version 5.5 of CMS and Earlier Versions

If you compile and link your programs using the SAS/C Compiler under Version 5.5 or later of CMS, your program cannot be run under an earlier version of CMS. If you compile and link under an earlier version of CMS, your program will run under Version 5.5 or later.

If you are not sure what version of CMS you are using, look at the linkage map for the final link-edit of your program. If you see L$CMAINC as a CSECT, you're running under a version prior to 5.5. If you see L$CMAINB instead, you're using Version 5.5 or later.

Shared Segments

Check with the SAS/C systems programmer at your site to determine if all of the following conditions occur:

- the SAS/C run-time routines are installed in segments
- the SAS System is installed in segments
- you are running under a version of CMS prior to Version 5.5
- the segments for the run-time libraries and the SAS System overlap.

If all of these conditions are true at your site, you must do one of the following:

- have your systems programmer de-install the SAS/C run-time library as segments. (This is the preferrable solution).
- zap your procedure module so that the SAS/C run-time routines are dynamically loaded instead of being fetched from segments. If this is necessary, have your systems programmer work with SAS Institute's Technical Support staff.

Compiling and Link-editing FORTRAN Programs

This section illustrates sample REXX EXECs provided with SAS/TOOLKIT software to compile and link-edit procedures and IFFCs with the VS FORTRAN Version 2 Compiler. Note that all of the EXECs illustrated in this section include a special step to run a FORTRAN preprocessor, invoked with the FTNPREP EXEC. This preprocessor ensures that the FORTRAN statements in your source code are in a format acceptable to the compiler on your operating system. The FTNPREP EXEC then runs the Fortran Version 2 Compiler to compile the preprocessed source code.

All of the EXECs used to compile and link FORTRAN programs use the FTNMAC MACLIB for the FORTRAN macro library. Your site may have this library installed under a different name. In that case, you need to modify the EXECs provided with SAS/TOOLKIT software to use the correct name at your site.

The following sections also list the contents of the TEXT file needed by the linkage editor.

FORTRAN Procedures

This section describes the TEXT file needed to link FORTRAN procedures and illustrates the PRCJOBF2 EXEC for compiling and linking.

Required TEXT File

Create a TEXT file to specify input to the linkage editor. The name of this file appears as the first operand when you execute the LKEDPV6 EXEC (see the last statement in the EXEC listed in the next section). This file contains the following statements:

```
INCLUDE program-constants
INCLUDE text1
...
INCLUDE textn
INCLUDE SYSLIB(PRCINT)
ENTRY   ENTRY
NAME    module(R)
```

where *program-constants* is the name of the file created by the PROC USERPROC step. This name should match *module* in the NAME statement, and both names should be the same as the value of the MODNAME= argument in the PROC USERPROC statement. Note that this must be the first line in the TEXT file. The *text1* through *textn* files are all of the compiled modules of your procedure; that is, they are the TEXT files created by the compiler. For procedures, you need to specify both the compiled procedure code and the compiled grammar function. The PRCINT module is a required part of SAS/TOOLKIT software.

The PRCJOBF2 EXEC

This REXX EXEC illustrates how to compile and link-edit a procedure with the FORTRAN compiler. This sample EXEC uses the following names for files:

EXAMPLE	is the name of the executable file created by the linkage editor.
EXAMPLEF	is the name of the file containing the source code to be compiled.
EXMPLG	is the name your procedure uses to call the grammar function.
EXMPLGF	is the name of the grammar function created by the USERPROC procedure.
EXMPLNKF	is the name of the TEXT file containing the statements for the linkage editor.

You can change any of these names, but be sure that all occurrences of the name are changed.

Listing A1.5
PRCJOBF2 EXEC

```
/* REXX EXEC to compile and linkedit the EXAMPLE procedure in FORTRAN */
trace all

/*-----create SAS program to build program constants object-----*/
queue 'PROC USERPROC OBJECT=''EXAMPLE TEXT'' MODNAME=EXAMPLE'
queue '     MODTYPE=PROC;'

/*-----create SAS program to build grammar function in SAS/C----*/
queue 'PROC USERPROC GRAMMAR=''EXAMPLE GRM'' GRAMFUNC=EXMPLG'
queue '     FUNCFILE=''EXMPLGF FORTRAN'' NOPRINT'
queue '     GRAMLANG=FORTRAN INCLLIB=''GRMMAC MACLIB *''; RUN;'
queue

/*-----copy SAS program to TEMP SAS temporary file-----*/
'EXECIO * DISKW TEMP SAS A3 1 F 80 (FINIS'

/*-----invoke the SAS System to create these entities-----*/
'EXEC SAS6 TEMP (NOLOG)'

/*-----necessary global for compilation of PL/I toolkit items-----*/
'GLOBAL MACLIB FTNMAC'

/*-----preprocess/compile the procedure source-----*/
'EXEC FTNPREP EXAMPLEF'
'RENAME TEMP LISTING A EXAMPLEF LISTING A'
/*-----do likewise for grammar function-----*/
'EXEC FTNPREP EXMPLGF'

/*-----housekeeping-----*/
'ERASE TEMP FORTRAN'

/*-----request linkedit to create EXAMPLE LOADLIB-----*/
'FILEDEF EXAMPLE DISK EXAMPLE TEXT * (PERM'
'FILEDEF EXAMPLEF DISK EXAMPLEF TEXT * (PERM'
'FILEDEF EXMPLGF  DISK EXMPLGF  TEXT * (PERM'
'EXEC LKEDPV6 EXMPLNKF EXAMPLE FORTRAN PROC'

exit
```

FORTRAN Functions and CALL Routines

This section describes the TEXT file needed to link FORTRAN functions and CALL routines and illustrates the IFFJOBUF EXEC for compiling and linking.

Required TEXT File

Create a TEXT file to specify input to the linkage editor. The name of this file appears as the first operand when you execute the LKEDPV6 EXEC (see the last statement in the IFFJOBUF EXEC listed in the next section). This file contains the following statements:

```
INCLUDE program-constants
INCLUDE text
INCLUDE SYSLIB(IFFEXTF)
INCLUDE SYSLIB(IFFINT)
INCLUDE SYSLIB(FTNFACE)
INCLUDE SYSLIB(PRCINT)
ENTRY   IFFFTN
NAME    module(R)
```

where *program-constants* is the name of the file created by the PROC USERPROC step. This name should match *module* in the NAME statement, and both names should be the same as the value of the MODNAME= argument in the PROC

USERPROC statement. Note that this must be the first line in the TEXT file. The *text* file is the compiled function; that is, *text* is the TEXT file created by the FORTRAN Compiler. The IFFEXTF module is needed only if you do not include an IFFEXT routine in your function source code. The IFFINT and FTNFACE modules are required parts of SAS/TOOLKIT software. You need to specify PRCINT only when your function uses SAS_X or SAS_Z routines.

The IFFJOBUF EXEC

This REXX EXEC illustrates how to compile and link-edit a function or CALL routine with the FORTRAN compiler. This sample EXEC uses the following names for files:

FUNCF — is the name of the file containing the source code to be compiled.

FUNCLNKF — is the name of the TEXT file containing the statements for the linkage editor.

IFFEXTF — is the name of a dummy exit routine that you need to compile and link with your function only if you do not code an IFFEXT routine in your function.

UWUTESTF — is the name of the program constants object file and the executable file created by the linkage editor.

You can change any of these names, but be sure that all occurrences of the name are changed.

Listing A1.6
IFFJOBUF EXEC

```
/* REXX EXEC to compile and linkedit the sample functions in FORTRAN */
trace all

/*-----create SAS program to build program constants object-----*/
queue 'PROC USERPROC OBJECT=''UWUTESTF TEXT'' MODNAME=UWUTESTF'
queue '     MODTYPE=FUNCTION;'
queue

/*-----copy SAS program to TEMP SAS temporary file-----*/
'EXECIO * DISKW TEMP SAS A3 1 F 80 (FINIS'

/*-----invoke the SAS System to create these entities-----*/
'EXEC SAS6 TEMP (NOLOG)'

/*-----necessary global for compilation of PL/I toolkit items-----*/
'GLOBAL MACLIB FTNMAC'

/*-----preprocess/compile the function source-----*/
'EXEC FTNPREP FUNCF'

/*-----COMPILATION OF DUMMY IFFEXT SOURCE-----*/
'FORTVS2 IFFEXTF'

/*-----housekeeping-----*/
'ERASE TEMP FORTRAN'

/*-----request linkedit to create UWUTESTF LOADLIB-----*/
'FILEDEF UWUTESTF DISK UWUTESTF TEXT * (PERM'
'FILEDEF FUNCF    DISK FUNCF    TEXT * (PERM'
'FILEDEF IFFEXTF  DISK IFFEXTF  TEXT * (PERM'
'EXEC LKEDPV6 FUNCLNKF UWUTESTF FORTRAN IFF'

exit
```

FORTRAN Informats

This section describes the TEXT file needed to link FORTRAN informats and illustrates the IFFJOBIF EXEC for compiling and linking.

Required TEXT File

Create a TEXT file to specify input to the linkage editor. The name of this file appears as the first operand when you execute the LKEDPV6 EXEC (see the last statement in the IFFJOBIF EXEC listed in the next section). This file contains the following statements:

```
INCLUDE program-constants
INCLUDE text
INCLUDE SYSLIB(IFFEXTF)
INCLUDE SYSLIB(IFFINT)
INCLUDE SYSLIB(FTNFACE)
INCLUDE SYSLIB(PRCINT)
ENTRY   IFFFTN
NAME    module(R)
```

where *program-constants* is the name of the file created by the PROC USERPROC step. This name should match *module* in the NAME statement, and both names should be the same as the value of the MODNAME= argument in the PROC USERPROC statement. Note that this must be the first line in the TEXT file. The *text* file is the compiled informat; that is, *text* is the TEXT file created by the FORTRAN compiler. The IFFEXTF module is needed only if you do not include an IFFEXT routine in your informat source code. The IFFINT and FTNFACE modules are required parts of SAS/TOOLKIT software. You need to specify PRCINT only when your informat uses SAS_X or SAS_Z routines.

The IFFJOBIF EXEC

This REXX EXEC illustrates how to compile and link-edit an informat with the VS FORTRAN Version 2 Compiler. This sample EXEC uses the following names for files:

IFFEXTF
: is the name of a dummy exit routine that you need to compile and link with your informat only if you do not code an IFFEXT routine in your informat.

INFMLNKF
: is the name of the TEXT file containing the statements for the linkage editor.

INFMTF
: is the name of the file containing the source code to be compiled.

UWITESTI
: is the name of the program constants object file and the executable file created by the linkage editor.

You can change any of these names, but be sure that all occurrences of the name are changed.

Listing A1.7
IFFJOBIF EXEC

```
/* REXX EXEC to compile and linkedit the sample informats in FORTRAN */
trace all

/*-----create SAS program to build program constants object-----*/
queue 'PROC USERPROC OBJECT=''UWITESTI TEXT'' MODNAME=UWITESTI'
queue '      MODTYPE=INFORMAT;'
queue

/*-----copy SAS program to TEMP SAS temporary file-----*/
'EXECIO * DISKW TEMP SAS A3 1 F 80 (FINIS'

/*-----invoke the SAS System to create these entities-----*/
'EXEC SAS6 TEMP (NOLOG)'

'GLOBAL MACLIB FTNMAC'
'EXEC FTNPREP INFMTF'
'FORTVS2 IFFEXTF'

/*-----request linkedit to create UWITESTI LOADLIB-----*/
'FILEDEF UWITESTI DISK UWITESTI TEXT * (PERM'
'FILEDEF INFMTF   DISK INFMTF   TEXT * (PERM'
'FILEDEF IFFEXTF  DISK IFFEXTF  TEXT * (PERM'
'EXEC LKEDPV6 INFMLNKF UWITESTI FORTRAN IFF'

exit
```

FORTRAN Formats

This section describes the TEXT file needed to link FORTRAN formats and illustrates the IFFJOBFF EXEC for compiling and linking.

Required TEXT File

Create a TEXT file to specify input to the linkage editor. The name of this file appears as the first operand when you execute the LKEDPV6 EXEC (see the last statement in the IFFJOBFF EXEC listed in the next section). This file contains the following statements:

```
INCLUDE program-constants
INCLUDE text
INCLUDE SYSLIB(IFFEXTF)
INCLUDE SYSLIB(IFFINT)
INCLUDE SYSLIB(FTNFACE)
INCLUDE SYSLIB(PRCINT)
ENTRY   IFFFTN
NAME    module(R)
```

where *program-constants* is the name of the file created by the PROC USERPROC step. This name should match *module* in the NAME statement, and both names should be the same as the value of the MODNAME= argument in the PROC USERPROC statement. Note that this must be the first line in the TEXT file. The *text* file is the compiled format; that is, *text* is the TEXT file created by the FORTRAN compiler. The IFFEXTF module is needed only if you do not include an IFFEXT routine in your format source code. The IFFINT and FTNFACE modules are required parts of SAS/TOOLKIT software. You need to specify PRCINT only when your format uses SAS_X or SAS_Z routines.

The IFFJOBFF EXEC

This REXX EXEC illustrates how to compile and link-edit a format with the VS FORTRAN Version 2 Compiler. This sample EXEC uses the following names for files:

FMTF
: is the name of the file containing the source code to be compiled.

FMTLNKF
: is the name of the TEXT file containing the statements for the linkage editor.

IFFEXTF
: is the name of a dummy exit routine that you need to compile and link with your format only if you do not code an IFFEXT routine in your format.

UWFONEOF
: is the name of the program constants object file and the executable file created by the linkage editor.

You can change any of these names, but be sure that all occurrences of the name are changed.

Listing A1.8
IFFJOBFF EXEC

```
/* REXX EXEC to compile and linkedit the sample formats in FORTRAN */
trace all

/*-----create SAS program to build program constants object-----*/
queue 'PROC USERPROC OBJECT=''UWFONEOF TEXT'' MODNAME=UWFONEOF'
queue '      MODTYPE=FORMAT;'
queue

/*-----copy SAS program to TEMP SAS temporary file-----*/
'EXECIO * DISKW TEMP SAS A3 1 F 80 (FINIS'

/*-----invoke the SAS System to create these entities-----*/
'EXEC SAS6 TEMP (NOLOG)'

'GLOBAL MACLIB FTNMAC'
'EXEC FTNPREP FMTF'
'FORTVS2 IFFEXTF'

/*-----request linkedit to create UWFONEOF LOADLIB-----*/
'FILEDEF UWFONEOF DISK UWFONEOF TEXT * (PERM'
'FILEDEF FMTF     DISK FMTF     TEXT * (PERM'
'FILEDEF IFFEXTF  DISK IFFEXTF  TEXT * (PERM'
'EXEC LKEDPV6 FMTLNKF  UWFONEOF FORTRAN IFF'

exit
```

Compiling and Link-editing PL/I Programs

This section illustrates sample REXX EXECs provided with SAS/TOOLKIT software to compile and link-edit procedures and IFFCs with the PL/I Optimizing Compiler.

All of the EXECs used to compile and link PL/I programs use the PLIMAC MACLIB for the PL/I macro library. Your site may have this library installed under a different name. In that case, you need to modify the EXECs provided with SAS/TOOLKIT software to use the correct name at your site.

The following sections also list the contents of the TEXT file needed by the linkage editor.

PL/I Procedures

This section describes the TEXT file needed to link PL/I procedures and illustrates the PRCJOBP2 EXEC for compiling and linking.

Required TEXT File

Create a TEXT file to specify input to the linkage editor. The name of this file appears as the first operand when you execute the LKEDPV6 EXEC (see the last statement in the EXEC listed in the next section). This file contains the following statements:

```
INCLUDE program-constants
INCLUDE text1
...
INCLUDE textn
INCLUDE SYSLIB(PRCINT)
ENTRY   ENTRY
NAME    module(R)
```

where *program-constants* is the name of the file created by the PROC USERPROC step. This name should match *module* in the NAME statement, and both names should be the same as the value of the MODNAME= argument in the PROC USERPROC statement. Note that this must be the first line in the TEXT file. The *text1* through *textn* files are all of the compiled modules of your procedure; that is, they are the TEXT files created by the compiler. For procedures, you need to specify both the compiled procedure code and the compiled grammar function. The PRCINT module is a required part of SAS/TOOLKIT software.

The PRCJOBP2 EXEC

This REXX EXEC illustrates how to compile and link-edit a procedure with the PL/I Optimizing Compiler. This sample EXEC uses the following names for files:

EXAMPLE	is the name of the executable file created by the linkage editor.
EXAMPLEP	is the name of the file containing the source code to be compiled.
EXMPLG	is the name your procedure uses to call the grammar function.
EXMPLGP	is the name of the grammar function created by the USERPROC procedure.
EXMPLNKP	is the name of the TEXT file containing the statements for the linkage editor.

You can change any of these names, but be sure that all occurrences of the name are changed.

Listing A1.9
PRCJOBP2 EXEC

```
/* REXX EXEC to compile and linkedit the EXAMPLE procedure in PL/I */

/*-----create SAS program to build program constants object-----*/
queue 'PROC USERPROC OBJECT=''EXAMPLE TEXT'' MODNAME=EXAMPLE'
queue '     MODTYPE=PROC;'

/*-----create SAS program to build grammar function in SAS/C----*/
queue 'PROC USERPROC GRAMMAR=''EXAMPLE GRM'' GRAMFUNC=EXMPLG'
queue '     FUNCFILE=''EXMPLGP PLIOPT'' NOPRINT'
queue '     GRAMLANG=PLI INCLLIB=''GRMMAC MACLIB *''; RUN;'
queue

/*-----copy SAS program to TEMP SAS temporary file-----*/
'EXECIO * DISKW TEMP SAS A3 1 F 80 (FINIS'

/*-----invoke the SAS System to create these entities-----*/
'EXEC SAS6 TEMP (NOLOG)'

/*-----necessary global for compilation of PL/I toolkit items-----*/
'GLOBAL MACLIB PLIMAC'

/*-----compile the procedure and grammar functions-----*/
'PLI EXAMPLEP (MACRO INCLUDE)'
'PLI EXMPLGP (MACRO INCLUDE)'

/*-----request linkedit to create EXAMPLE LOADLIB-----*/
'FILEDEF EXAMPLE DISK EXAMPLE TEXT * (PERM'
'FILEDEF EXAMPLEP DISK EXAMPLEP TEXT * (PERM'
'FILEDEF EXMPLGP  DISK EXMPLGP  TEXT * (PERM'
'EXEC LKEDPV6 EXMPLNKP EXAMPLE PLI PROC'

exit
```

PL/I Functions and CALL Routines

This section describes the TEXT file needed to link PL/I functions and CALL routines and illustrates the IFFJOBUP EXEC for compiling and linking.

Required TEXT File

Create a TEXT file to specify input to the linkage editor. The name of this file appears as the first operand when you execute the LKEDPV6 EXEC (see the last statement in the IFFJOBUP EXEC listed in the next section). This file contains the following statements:

```
INCLUDE program-constants
INCLUDE text
INCLUDE SYSLIB(IFFEXTP)
INCLUDE SYSLIB(IFFINT)
INCLUDE SYSLIB(PLIFACE)
INCLUDE SYSLIB(PRCINT)
ENTRY   IFFPLI
NAME    module(R)
```

where *program-constants* is the name of the file created by the PROC USERPROC step. This name should match *module* in the NAME statement, and both names should be the same as the value of the MODNAME= argument in the PROC USERPROC statement. Note that this must be the first line in the TEXT file. The *text* file is the compiled function; that is, *text* is the TEXT file created by the PL/I compiler. The IFFEXTP module is needed only if you do not include an IFFEXT routine in your function source code. The IFFINT and PLIFACE modules are required parts of SAS/TOOLKIT software. You need to specify PRCINT only when your function uses SAS_X or SAS_Z routines.

The IFFJOBUP EXEC

This REXX EXEC illustrates how to compile and link-edit a function or CALL routine with the PL/I Optimizing Compiler. This sample EXEC uses the following names for files:

FUNCLNKP
: is the name of the TEXT file containing the statements for the linkage editor.

FUNCP1-FUNCP6
: are the names of the file containing the source code to be compiled.

IFFEXTP
: is the name of a dummy exit routine that you need to compile and link with your function only if you do not code an IFFEXT routine in your function.

UWUTESTF
: is the name of the program constants object file and the executable file created by the linkage editor.

You can change any of these names, but be sure that all occurrences of the name are changed.

Listing A1.10
IFFJOBUP EXEC

```
/* REXX EXEC to compile and linkedit the sample functions in PL/I */
trace all

/*-----create SAS program to build program constants object-----*/
queue 'PROC USERPROC OBJECT=''UWUTESTF TEXT'' MODNAME=UWUTESTF'
queue '     MODTYPE=FUNCTION;'
queue

/*-----copy SAS program to TEMP SAS temporary file-----*/
'EXECIO * DISKW TEMP SAS A3 1 F 80 (FINIS'

/*-----invoke the SAS System to create these entities-----*/
'EXEC SAS6 TEMP (NOLOG)'

/*-----necessary global for compilation of PL/I toolkit items-----*/
'GLOBAL MACLIB PLIMAC'

'PLI FUNCP1 (MACRO INCLUDE)'
'PLI FUNCP2 (MACRO INCLUDE)'
'PLI FUNCP3 (MACRO INCLUDE)'
'PLI FUNCP4 (MACRO INCLUDE)'
'PLI FUNCP5 (MACRO INCLUDE)'
'PLI FUNCP6 (MACRO INCLUDE)'
'PLI IFFEXTP (MACRO INCLUDE)'

/*-----request linkedit to create UWUTESTF LOADLIB-----*/
'FILEDEF UWUTESTF DISK UWUTESTF TEXT * (PERM'
'FILEDEF FUNCP1   DISK FUNCP1   TEXT * (PERM'
'FILEDEF FUNCP2   DISK FUNCP2   TEXT * (PERM'
'FILEDEF FUNCP3   DISK FUNCP3   TEXT * (PERM'
'FILEDEF FUNCP4   DISK FUNCP4   TEXT * (PERM'
'FILEDEF FUNCP5   DISK FUNCP5   TEXT * (PERM'
'FILEDEF FUNCP6   DISK FUNCP6   TEXT * (PERM'
'FILEDEF IFFEXTP  DISK IFFEXTP  TEXT * (PERM'
'EXEC LKEDPV6 FUNCLNKP UWUTESTF PLI IFF'

exit
```

PL/I Informats

This section describes the TEXT file needed to link PL/I informats and illustrates the IFFJOBIP EXEC for compiling and linking.

Required TEXT File

Create a TEXT file to specify input to the linkage editor. The name of this file appears as the first operand when you execute the LKEDPV6 EXEC (see the last statement in the IFFJOBIP EXEC listed in the next section). This file contains the following statements:

```
INCLUDE program-constants
INCLUDE text
INCLUDE SYSLIB(IFFEXTP)
INCLUDE SYSLIB(IFFINT)
INCLUDE SYSLIB(PLIFACE)
INCLUDE SYSLIB(PRCINT)
ENTRY   IFFPLI
NAME    module(R)
```

where *program-constants* is the name of the file created by the PROC USERPROC step. This name should match *module* in the NAME statement, and both names should be the same as the value of the MODNAME= argument in the PROC USERPROC statement. Note that this must be the first line in the TEXT file. The *text* file is the compiled informat; that is, *text* is the TEXT file created by the PL/I compiler. The IFFEXTP module is needed only if you do not include an IFFEXT routine in your informat source code. The IFFINT and PLIFACE modules are required parts of SAS/TOOLKIT software. You need to specify PRCINT only when your informat uses SAS_X or SAS_Z routines.

The IFFJOBIP EXEC

This REXX EXEC illustrates how to compile and link-edit an informat with the PL/I Optimizing Compiler. This sample EXEC uses the following names for files:

IFFEXTP	is the name of a dummy exit routine that you need to compile and link with your informat only if you do not code an IFFEXT routine in your informat.
INFMLNKP	is the name of the TEXT file containing the statements for the linkage editor.
INFMTP1-INFMTP3	are the names of the file containing the source code to be compiled.
UWITESTI	is the name of the program constants object file and the executable file created by the linkage editor.

You can change any of these names, but be sure that all occurrences of the name are changed.

Listing A1.11
IFFJOBIP EXEC

```
 /* REXX EXEC to compile and linkedit the sample informats in PL/I */
trace all

 /*-----create SAS program to build program constants object-----*/
queue 'PROC USERPROC OBJECT=''UWITESTI TEXT'' MODNAME=UWITESTI'
queue '     MODTYPE=INFORMAT;'
queue

 /*-----copy SAS program to TEMP SAS temporary file-----*/
'EXECIO * DISKW TEMP SAS A3 1 F 80 (FINIS'

 /*-----invoke the SAS System to create these entities-----*/
'EXEC SAS6 TEMP (NOLOG)'

 /*-----necessary global for compilation of PL/I toolkit items-----*/
'GLOBAL MACLIB PLIMAC'

'PLI INFMTP1 (MACRO INCLUDE)'
'PLI INFMTP2 (MACRO INCLUDE)'
'PLI INFMTP3 (MACRO INCLUDE)'
'PLI IFFEXTP (MACRO INCLUDE)'

 /*-----request linkedit to create UWITESTI LOADLIB-----*/
'FILEDEF UWITESTI DISK UWITESTI TEXT * (PERM'
'FILEDEF INFMTP1  DISK INFMTP1  TEXT * (PERM'
'FILEDEF INFMTP2  DISK INFMTP2  TEXT * (PERM'
'FILEDEF INFMTP3  DISK INFMTP3  TEXT * (PERM'
'FILEDEF IFFEXTP  DISK IFFEXTP  TEXT * (PERM'
'EXEC LKEDPV6 INFMLNKP UWITESTI PLI IFF'

 exit
```

PL/I Formats

This section describes the TEXT file needed to link PL/I formats and illustrates the IFFJOBFP EXEC for compiling and linking.

Required TEXT File

Create a TEXT file to specify input to the linkage editor. The name of this file appears as the first operand when you execute the LKEDPV6 EXEC (see the last statement in the IFFJOBFP EXEC listed in the next section). This file contains the following statements:

```
INCLUDE program-constants
INCLUDE text
INCLUDE SYSLIB(IFFEXTP)
INCLUDE SYSLIB(IFFINT)
INCLUDE SYSLIB(PLIFACE)
INCLUDE SYSLIB(PRCINT)
ENTRY   IFFPLI
NAME    module(R)
```

where *program-constants* is the name of the file created by the PROC USERPROC step. This name should match *module* in the NAME statement, and both names should be the same as the value of the MODNAME= argument in the PROC USERPROC statement. Note that this must be the first line in the TEXT file. The *text* file is the compiled format; that is, *text* is the TEXT file created by the PL/I compiler. The IFFEXTP module is needed only if you do not include an IFFEXT routine in your format source code. The IFFINT and PLIFACE modules are required parts of SAS/TOOLKIT software. You need to specify PRCINT only when your format uses SAS_X or SAS_Z routines.

The IFFJOBFP EXEC

This REXX EXEC illustrates how to compile and link-edit a format with the PL/I Optimizing Compiler. This sample EXEC uses the following names for files:

FMTLNKP
: is the name of the TEXT file containing the statements for the linkage editor.

FMTP1-FMTP3
: are the names of the file containing the source code to be compiled.

IFFEXTP
: is the name of a dummy exit routine that you need to compile and link with your format only if you do not code an IFFEXT routine in your format.

UWFONEOF
: is the name of the program constants object file and the executable file created by the linkage editor.

You can change any of these names, but be sure that all occurrences of the name are changed.

Listing A1.12
IFFJOBFP EXEC

```
/* REXX EXEC to compile and linkedit the sample formats in PL/I */
trace all

 /*-----create SAS program to build program constants object-----*/
queue 'PROC USERPROC OBJECT=''UWFONEOF TEXT'' MODNAME=UWFONEOF'
queue '     MODTYPE=FORMAT;'
queue

 /*-----copy SAS program to TEMP SAS temporary file-----*/
'EXECIO * DISKW TEMP SAS A3 1 F 80 (FINIS'

 /*-----invoke the SAS System to create these entities-----*/
'EXEC SAS6 TEMP (NOLOG)'

 /*-----necessary global for compilation of PL/I toolkit items-----*/
'GLOBAL MACLIB PLIMAC'

'PLI FMTP1 (MACRO INCLUDE)'
'PLI FMTP2 (MACRO INCLUDE)'
'PLI FMTP3 (MACRO INCLUDE)'
'PLI IFFEXTP (MACRO INCLUDE)'

 /*-----request linkedit to create UWFONEOF LOADLIB-----*/
'FILEDEF UWFONEOF DISK UWFONEOF TEXT * (PERM'
'FILEDEF FMTP1    DISK FMTP1    TEXT * (PERM'
'FILEDEF FMTP2    DISK FMTP2    TEXT * (PERM'
'FILEDEF FMTP3    DISK FMTP3    TEXT * (PERM'
'FILEDEF IFFEXTP  DISK IFFEXTP  TEXT * (PERM'
'EXEC LKEDPV6 FMTLNKP UWFONEOF PLI IFF'

exit
```

Assembling and Link-editing IBM 370 Assembler Programs

This section illustrates sample REXX EXECs provided with SAS/TOOLKIT software to assemble and link-edit procedures and IFFCs with the IBM 370 Version 2 H-level Assembler.

All of the EXECs used to assemble and link assembler programs use the ASMMAC MACLIB for the assembler macro library. Your site may have this library installed under a different name. In that case, you need to modify the EXECs provided with SAS/TOOLKIT software to use the correct name at your site.

The following sections also list the contents of the TEXT file needed by the linkage editor.

Assembler Procedures

This section describes the TEXT file needed to link assembler procedures and illustrates the PRCJOBA2 EXEC for assembling and linking.

Required TEXT File

Create a TEXT file to specify input to the linkage editor. The name of this file appears as the first operand when you execute the LKEDPV6 EXEC (see the last statement in the EXEC listed in the next section). This file contains the following statements:

```
INCLUDE program-constants
INCLUDE text1
...
INCLUDE textn
INCLUDE SYSLIB(PRCINT)
ENTRY   ENTRY
NAME    module(R)
```

where *program-constants* is the name of the file created by the PROC USERPROC step. This name should match *module* in the NAME statement, and both names should be the same as the value of the MODNAME= argument in the PROC USERPROC statement. Note that this must be the first line in the TEXT file. The *text1* through *textn* files are all of the assembled modules of your procedure; that is, they are the TEXT files created by the assembler. For procedures, you need to specify both the assembled procedure code and the assembled grammar function. The PRCINT module is a required part of SAS/TOOLKIT software.

The PRCJOBA2 EXEC

This REXX EXEC illustrates how to assemble and link-edit a procedure with the IBM H-level 370 Assembler. This sample EXEC uses the following names for files:

EXAMPLE	is the name of the executable file created by the linkage editor.
EXAMPLEA	is the name of the file containing the source code to be assembled.
EXMPLG	is the name your procedure uses to call the grammar function.
EXMPLGA	is the name of the grammar function created by the USERPROC procedure.
EXMPLNKA	is the name of the TEXT file containing the statements for the linkage editor.

You can change any of these names, but be sure that all occurrences of the name are changed.

Listing A1.13
PRCJOBA2 EXEC

```
/* REXX EXEC to assemble and linkedit the EXAMPLE procedure in ASM */
/* ensure you're using V2 of HASM */

/*-----create SAS program to build program constants object-----*/
queue 'PROC USERPROC OBJECT=''EXAMPLE TEXT'' MODNAME=EXAMPLE'
queue '     MODTYPE=PROC;'

/*-----create SAS program to build grammar function in SAS/C----*/
queue 'PROC USERPROC GRAMMAR=''EXAMPLE GRM'' GRAMFUNC=EXMPLG'
queue '     FUNCFILE=''EXMPLGA ASSEMBLE'' NOPRINT'
queue '     GRAMLANG=ASM370 INCLLIB=''GRMMAC MACLIB *''; RUN;'
queue

/*-----copy SAS program to TEMP SAS temporary file-----*/
'EXECIO * DISKW TEMP SAS A3 1 F 80 (FINIS'

/*-----invoke the SAS System to create these entities-----*/
'EXEC SAS6 TEMP (NOLOG)'

/*-----necessary global for assembly of toolkit items-----*/
'GLOBAL MACLIB ASMMAC'

/*-----assemble the procedure and grammar functions-----*/
'HASM EXAMPLEA'
'HASM EXMPLGA'

/*-----request linkedit to create EXAMPLE LOADLIB-----*/
'FILEDEF EXAMPLE DISK EXAMPLE TEXT * (PERM'
'FILEDEF EXAMPLEA DISK EXAMPLEA TEXT * (PERM'
'FILEDEF EXMPLGA  DISK EXMPLGA  TEXT * (PERM'
'EXEC LKEDPV6 EXMPLNKA EXAMPLE ASM'

exit
```

Assembler Functions and CALL Routines

This section describes the TEXT file needed to link assembler functions and CALL routines and illustrates the IFFJOBUA EXEC for assembling and linking.

Required TEXT File

Create a TEXT file to specify input to the linkage editor. The name of this file appears as the first operand when you execute the LKEDPV6 EXEC (see the last statement in the IFFJOBUA EXEC listed in the next section). This file contains the following statements:

```
INCLUDE program-constants
INCLUDE text
INCLUDE SYSLIB(IFFEXTA)
INCLUDE SYSLIB(IFFINT)
INCLUDE SYSLIB(ASMFACE)
INCLUDE SYSLIB(PRCINT)
ENTRY   IFFASM
NAME    module(R)
```

where *program-constants* is the name of the file created by the PROC USERPROC step. This name should match *module* in the NAME statement, and both names should be the same as the value of the MODNAME= argument in the PROC USERPROC statement. Note that this must be the first line in the TEXT file. The *text* file is the assembled function; that is, *text* is the TEXT file created by the IBM H-level 370 Assembler. The IFFEXTA module is needed only if you do not include an IFFEXT routine in your function source code. The IFFINT and ASMFACE modules are required parts of SAS/TOOLKIT software. You need to specify PRCINT only when your function uses SAS_X or SAS_Z routines.

The IFFJOBUA EXEC

This REXX EXEC illustrates how to assemble and link-edit a function or CALL routine with the IBM H-level 370 Assembler. This sample EXEC uses the following names for files:

FUNCA — is the name of the file containing the source code to be assembled.

FUNCLNKA — is the name of the TEXT file containing the statements for the linkage editor.

UWUTESTF — is the name of the program constants object file and the executable file created by the linkage editor.

You can change any of these names, but be sure that all occurrences of the name are changed.

Listing A1.14
IFFJOBUA EXEC

```
/* REXX EXEC to assemble and linkedit the sample functions in ASM */
/* ensure you're using V2 of HASM */

/*-----create SAS program to build program constants object-----*/
queue 'PROC USERPROC OBJECT=''UWUTESTF TEXT'' MODNAME=UWUTESTF'
queue '     MODTYPE=FUNCTION;'
queue

/*-----copy SAS program to TEMP SAS temporary file-----*/
'EXECIO * DISKW TEMP SAS A3 1 F 80 (FINIS'

/*-----invoke the SAS System to create these entities-----*/
'EXEC SAS6 TEMP (NOLOG)'

/*-----necessary global for assembly of toolkit items-----*/
'GLOBAL MACLIB ASMMAC'

/*-----assemble the function code-----*/
'HASM FUNCA'

/*-----request linkedit to create UWUTESTF LOADLIB-----*/
'FILEDEF UWUTESTF DISK UWUTESTF TEXT * (PERM'
'FILEDEF FUNCA    DISK FUNCA    TEXT * (PERM'
'EXEC LKEDPV6 FUNCLNKA UWUTESTF ASM'

exit
```

Assembler Informats

This section describes the TEXT file needed to link assembler informats and illustrates the IFFJOBIA EXEC for assembling and linking.

Required TEXT File

Create a TEXT file to specify input to the linkage editor. The name of this file appears as the first operand when you execute the LKEDPV6 EXEC (see the last statement in the IFFJOBIA EXEC listed in the next section). This file contains the following statements:

```
INCLUDE program-constants
INCLUDE text
INCLUDE SYSLIB(IFFEXTA)
INCLUDE SYSLIB(IFFINT)
INCLUDE SYSLIB(ASMFACE)
INCLUDE SYSLIB(PRCINT)
ENTRY   IFFASM
NAME    module(R)
```

where *program-constants* is the name of the file created by the PROC USERPROC step. This name should match *module* in the NAME statement, and both names should be the same as the value of the MODNAME= argument in the PROC USERPROC statement. Note that this must be the first line in the TEXT file. The *text* file is the assembled informat; that is, *text* is the TEXT file created by the IBM H-level 370 Assembler. The IFFEXTA module is needed only if you do not include an IFFEXT routine in your informat source code. The IFFINT and ASMFACE modules are required parts of SAS/TOOLKIT software. You need to specify PRCINT only when your informat uses SAS_X or SAS_Z routines.

The IFFJOBIA EXEC

This REXX EXEC illustrates how to assemble and link-edit an informat with the IBM H-level 370 Assembler. This sample EXEC uses the following names for files:

INFMLNKA — is the name of the TEXT file containing the statements for the linkage editor.

INFMTA — is the name of the file containing the source code to be assembled.

UWITESTI — is the name of the program constants object file and the executable file created by the linkage editor.

You can change any of these names, but be sure that all occurrences of the name are changed.

Listing A1.15
IFFJOBIA EXEC

```
 /* REXX EXEC to assemble and linkedit the sample informats in ASM */
 /* ensure you're using V2 of HASM */

 /*-----create SAS program to build program constants object-----*/
queue 'PROC USERPROC OBJECT=''UWITESTI TEXT'' MODNAME=UWITESTI'
queue '     MODTYPE=INFORMAT;'
queue

 /*-----copy SAS program to TEMP SAS temporary file-----*/
'EXECIO * DISKW TEMP SAS A3 1 F 80 (FINIS'

 /*-----invoke the SAS System to create these entities-----*/
'EXEC SAS6 TEMP (NOLOG)'

 /*-----necessary global for assembly of toolkit items-----*/
'GLOBAL MACLIB ASMMAC'

 /*-----assemble the function code-----*/
'HASM INFMTA'

 /*-----request linkedit to create UWITESTI LOADLIB-----*/
'FILEDEF UWITESTI DISK UWITESTI TEXT * (PERM'
'FILEDEF INFMTA   DISK INFMTA   TEXT * (PERM'
'EXEC LKEDPV6 INFMLNKA UWITESTI ASM'

exit
```

Assembler Formats

This section describes the TEXT file needed to link assembler formats and illustrates the IFFJOBFA EXEC for assembling and linking.

Required TEXT File

Create a TEXT file to specify input to the linkage editor. The name of this file appears as the first operand when you execute the LKEDPV6 EXEC (see the last statement in the IFFJOBFA EXEC listed in the next section). This file contains the following statements:

```
INCLUDE program-constants
INCLUDE text
INCLUDE SYSLIB(IFFEXTA)
INCLUDE SYSLIB(IFFINT)
INCLUDE SYSLIB(ASMFACE)
INCLUDE SYSLIB(PRCINT)
ENTRY   IFFASM
NAME    module(R)
```

where *program-constants* is the name of the file created by the PROC USERPROC step. This name should match *module* in the NAME statement, and both names should be the same as the value of the MODNAME= argument in the PROC USERPROC statement. Note that this must be the first line in the TEXT file. The *text* file is the assembled format; that is, *text* is the TEXT file created by the IBM H-level 370 Assembler. The IFFEXTA module is needed only if you do not include an IFFEXT routine in your format source code. The IFFINT and ASMFACE modules are required parts of SAS/TOOLKIT software. You need to specify PRCINT only when your format uses SAS_X or SAS_Z routines.

The IFFJOBFA EXEC

This REXX EXEC illustrates how to assemble and link-edit a format with the IBM H-level 370 Assembler. This sample EXEC uses the following names for files:

FMTA
: is the name of the file containing the source code to be assembled.

FMTLNKA
: is the name of the TEXT file containing the statements for the linkage editor.

UWFONEOF
: is the name of the program constants object file and the executable file created by the linkage editor.

You can change any of these names, but be sure that all occurrences of the name are changed.

Listing A1.16
IFFJOBFA EXEC

```
 /* REXX EXEC to assemble and linkedit the sample formats in ASM */
 /* ensure you're using V2 of HASM */

 /*-----create SAS program to build program constants object-----*/
queue 'PROC USERPROC OBJECT=''UWFONEOF TEXT'' MODNAME=UWFONEOF'
queue '     MODTYPE=FORMAT;'
queue

 /*-----copy SAS program to TEMP SAS temporary file-----*/
'EXECIO * DISKW TEMP SAS A3 1 F 80 (FINIS'

 /*-----invoke the SAS System to create these entities-----*/
'EXEC SAS6 TEMP (NOLOG)'

 /*-----necessary global for assembly of toolkit items-----*/
'GLOBAL MACLIB ASMMAC'

 /*-----assemble the format code-----*/
'HASM FMTA'

 /*-----request linkedit to create UWFONEOF LOADLIB-----*/
'FILEDEF UWFONEOF DISK UWFONEOF TEXT * (PERM'
'FILEDEF FMTA     DISK FMTA     TEXT * (PERM'
'EXEC LKEDPV6 FMTLNKA  UWFONEOF ASM'

exit
```

Appendix 2 Using SAS/TOOLKIT™ Software under MVS

Introduction

This appendix provides sample JCL (job control language) for compiling and link-editing your procedure, function, CALL routine, informat, or format in various supported languages. Under MVS, you can use the following languages and compilers:

- the SAS/C Compiler
- the PL/I Optimizing Compiler
- the VS FORTRAN Version 2 Compiler
- the IBM 370 Assembler.

In addition, this appendix discusses how to use user-written SAS modules after they are compiled and how to make those modules available to other sites.

Accessing User-Written SAS Modules

This section discusses how to access user-written modules and how to make these modules available to other sites.

Specifying Where to Find Modules

After you have compiled and link-edited your program, you can access it by allocating the load library containing your program to the SASLIB DDname. You must also specify the library that contains the SAS/TOOLKIT load modules when you allocate the SASLIB DDname. If you are running under TSO, you must allocate this library before you invoke the SAS System on the SAS command. The following example illustrates how to allocate the SASLIB DDname under TSO:

```
ALLOC FI(SASLIB) DA('your-load-library' 'SAS/TOOLKIT-load-library') SHR REU
```

This example illustrates the JCL DD statement required for the SASLIB DDname:

```
//SASLIB  DD  DISP=SHR,DSN=your-load-library
//        DD  DISP=SHR,DSN=SAS/TOOLKIT-load-library
```

When you specify a SASLIB library, the SAS System searches this library for a load module before it searches the standard STEPLIB libraries. Note that you must allocate the SASLIB DDname before invoking the SAS System. You cannot use a FILENAME or LIBNAME statement or the TSO or X commands to allocate SASLIB from within a SAS session.

Distributing Modules to Other SAS Sites

If you distribute a module you have written to other SAS sites, you must also make available the SASPROCI module supplied with SAS/TOOLKIT software. You are free to redistribute SASPROCI, since it is treated as a transient module. If you do not supply SASPROCI to the other installation, your procedure or IFFC will not run at that site.

Note: You are *not* permitted to redistribute any other components of SAS/TOOLKIT software.

Compiling and Link-editing C Programs

This section illustrates sample JCL procedures provided with SAS/TOOLKIT software to compile and link-edit procedures and IFFCs with the SAS/C Compiler.

Following the sample JCL procedures are examples of how to use these procedures for each of the following:

- □ procedures
- □ functions or CALL routines
- □ informats
- □ formats.

All of the JCL procedures and the JCL used to invoke these procedures use the following sample PDS names:

TOOLKIT.SASC.*type*
: are your libraries. The types can include source (SRC), object (OBJ), and load (LOAD) libraries.

TOOLKIT.GRM
: is your grammar source module.

TOOLKIT.*type*
: are the SAS/TOOLKIT libraries.

SASC.LINKLIB
: is the SAS/C Compiler run-time library.

SASC.LOAD
: is the SAS/C Compiler load library.

▶ ***Caution*** .... ***Do not use a compiler option that globally causes data elements to be unaligned.*** SAS/TOOLKIT software expects the structures it uses to be aligned. If you want a structure used only by your program to be unaligned, you can use a compiler option that affects only that structure (if the compiler you are using provides such an option). For example, the __NOALIGNMEM option for the SAS/C Compiler can be used with a structure as follows:

```
__NOALIGNMEM struct ABC {
   structure declaration
   }
```

Consult the documentation for the compiler you are using to determine if an option similar to this one is available. ▲

Sample JCL Procedures for the SAS/C Compiler

SAS/TOOLKIT software provides the following three sample JCL procedures:

CCOMPILE	compiles a SAS module using the SAS/C Compiler.
CLINK	runs the CLINK module of the SAS/C Compiler.
UWLINK	runs the standard IBM linkage editor and links your SAS module with SAS/TOOLKIT software. If you are creating a SAS procedure, this step also links the grammar for your procedure with the main program.

You use these JCL procedures to compile and link-edit any SAS/TOOLKIT module written in the C language. Your SAS Software Representative customizes these JCL procedures at the time that SAS/TOOLKIT software is installed at your site. Ask your SAS Software Representative to help you locate these JCL procedures if you need to examine them before using them.

Note that the JCL illustrated in this section assumes that your site has installed the CCOMPILE, CLINK, and UWLINK procedures in a system procedure library at your site. If your site has not installed these procedures, ask your SAS Software Representative to provide you with the JCL included on the installation tape for SAS/TOOLKIT software. This JCL integrates the required procedures as instream procedures with the JCL illustrated in this section.

Note: A SAS/TOOLKIT load module can be loaded above the line with the MVS/XA operating system. This means that you should always use AMODE of 31 and RMODE of ANY. With those option choices, the module is loaded above the line if possible. If you are running under a non-XA system, the module is loaded below the line and everything functions normally. By default, the UWLINK procedure specifies AMODE=31 and RMODE=ANY in the statement that runs the IEWL program.

SAS/C Procedures

This JCL illustrates how to compile and link-edit a procedure with the SAS/C Compiler.

```
//*--------------OBTAIN THE VALIDATION OBJECT AND GRAMMAR FUNCTION---*
//MAKEOBJ  EXEC SAS6
//SASLIB     DD DSN=TOOLKIT.LOAD,DISP=SHR
//MCBOBJ     DD DSN=TOOLKIT.SASC.OBJ(EXAMPLE),DISP=SHR
//CTRANS     DD DSN=SASC.LOAD,DISP=SHR
//           DD DSN=SASC.LINKLIB,DISP=SHR
//SYSIN      DD *
PROC USERPROC OBJECT=MCBOBJ MODNAME=EXAMPLE MODTYPE=PROC;
PROC USERPROC GRAMMAR='TOOLKIT.GRM(EXAMPLE)' GRAMFUNC=EXMPLG
    FUNCFILE='TOOLKIT.SASC.SRC(EXMPLGC)'
    GRAMLANG=C INCLLIB='TOOLKIT.GRM'; RUN;
//*--------------COMPILE THE ITEM(S)---------------------------------*
//COMPILE  EXEC CCOMPILE,MEMBER=EXAMPLEC
//COMPILE  EXEC CCOMPILE,MEMBER=EXMPLGC
//*--------------CLINK THE ITEM(S)-----------------------------------*
//CLINK    EXEC CLINK
//SYSIN      DD *
INCLUDE COBJ(EXAMPLEC)          ALL C OBJECTS
INCLUDE COBJ(EXMPLGC)           ALL C OBJECTS
INCLUDE UWOBJ(PRCINT)           SUPPLIED BY SAS/TOOLKIT
//*--------------LINKEDIT THE PROCEDURE------------------------------*
//LINKEDIT EXEC UWLINK
//SYSLIN   DD  *
 INCLUDE  MCBOBJ(EXAMPLE)       GENERATED BY PROC USERPROC
 INCLUDE  CLINKOBJ              ALL LOCAL CLINK-ED CODE
 ENTRY    ENTRY                 INDICATES SAS PROCEDURE
 NAME     EXAMPLE(R)            NAME OF SAS PROCEDURE MODULE
//
```

SAS/C Functions and CALL Routines

This JCL illustrates how to compile and link-edit a function or CALL routine with the SAS/C Compiler.

Note that the statements in the CLINK step require that you specify three SAS/C Compiler modules, L$UMAIN, L$UEXIT, and L$UJUMP. A new version of the third module, L$UJUMP, is included in the SAS/TOOLKIT library. Be sure that the SYSLIB DD statement in the CLINK procedure specifies the SAS/TOOLKIT load library before the SAS/C Compiler libraries.

```
//*--------------OBTAIN THE VALIDATION OBJECT AND GRAMMAR FUNCTION---*
//MAKEOBJ  EXEC SAS6
//SASLIB     DD DSN=TOOLKIT.LOAD,DISP=SHR
//MCBOBJ     DD DSN=TOOLKIT.SASC.OBJ(UWUTESTF),DISP=SHR
//CTRANS     DD DSN=SASC.LOAD,DISP=SHR
//           DD DSN=SASC.LINKLIB,DISP=SHR
//SYSIN      DD *
PROC USERPROC OBJECT=MCBOBJ MODNAME=UWUTESTF MODTYPE=FUNCTION;
//*--------------COMPILE THE ITEM(S)---------------------------------*
//COMPILE  EXEC CCOMPILE,MEMBER=FUNCC
//COMPILE  EXEC CCOMPILE,MEMBER=IFFEXTC
//*--------------CLINK THE ITEM(S)-----------------------------------*
//CLINK    EXEC CLINK
//SYSIN      DD *
INCLUDE COBJ(FUNCC)             ALL C OBJECTS
INCLUDE COBJ(IFFEXTC)           ALL C OBJECTS
INCLUDE SYSLIB(L$UMAIN)         MAIN ROUTINE ENTRY NEEDED
INCLUDE SYSLIB(L$UEXIT)         EXIT ROUTINE ENTRY NEEDED
INCLUDE SYSLIB(L$UJUMP)         ENSURE BUG FIX INCLUDED
//*--------------LINKEDIT THE IFF------------------------------------*
//LINKEDIT EXEC UWLINK
//SYSLIN      DD  *
 INCLUDE  MCBOBJ(UWUTESTF)      GENERATED BY PROC USERPROC
 INCLUDE  CLINKOBJ              ALL LOCAL CLINK-ED CODE
 INCLUDE  UWOBJ(IFFINT)         SUPPLIED BY SAS/TOOLKIT
 INCLUDE  UWOBJ(SASCFACE)       SUPPLIED BY SAS/TOOLKIT
 INCLUDE  UWOBJ(PRCINT)         FOR SAS_ CALLS
 ENTRY    IFFSASC               INDICATES SAS/C ENTRY
 NAME     UWUTESTF(R)
//
//
```

SAS/C Informats

This JCL illustrates how to compile and link-edit an informat with the SAS/C Compiler.

```
//*--------------OBTAIN THE VALIDATION OBJECT AND GRAMMAR FUNCTION---*
//MAKEOBJ  EXEC SAS6
//SASLIB      DD DSN=TOOLKIT.LOAD,DISP=SHR
//MCBOBJ      DD DSN=TOOLKIT.SASC.OBJ(UWITESTI),DISP=SHR
//CTRANS      DD DSN=SASC.LOAD,DISP=SHR
//            DD DSN=SASC.LINKLIB,DISP=SHR
//SYSIN       DD *
PROC USERPROC OBJECT=MCBOBJ MODNAME=UWITESTI MODTYPE=INFORMAT;
//*--------------COMPILE THE ITEM(S)---------------------------------*
//COMPILE  EXEC CCOMPILE,MEMBER=INFMTC
//COMPILE  EXEC CCOMPILE,MEMBER=IFFEXTC
//*--------------CLINK THE ITEM(S)-----------------------------------*
//CLINK    EXEC CLINK
//SYSIN       DD *
INCLUDE COBJ(INFMTC)          ALL C OBJECTS
INCLUDE COBJ(IFFEXTC)         ALL C OBJECTS
INCLUDE SYSLIB(L$UMAIN)       MAIN ROUTINE ENTRY NEEDED
INCLUDE SYSLIB(L$UEXIT)       EXIT ROUTINE ENTRY NEEDED
INCLUDE SYSLIB(L$UJUMP)       ENSURE BUG FIX INCLUDED
//*--------------LINKEDIT THE IFF------------------------------------*
//LINKEDIT EXEC UWLINK
//SYSLIN       DD  *
 INCLUDE  MCBOBJ(UWITESTI)     GENERATED BY PROC USERPROC
 INCLUDE  CLINKOBJ             ALL LOCAL CLINK-ED CODE
 INCLUDE  UWOBJ(IFFINT)        SUPPLIED BY SAS/TOOLKIT
 INCLUDE  UWOBJ(SASCFACE)      SUPPLIED BY SAS/TOOLKIT
 INCLUDE  UWOBJ(PRCINT)        FOR SAS_ CALLS
 ENTRY    IFFSASC              INDICATES SAS/C ENTRY
 NAME     UWITESTI(R)
//
```

SAS/C Formats

This JCL illustrates how to compile and link-edit a format with the SAS/C Compiler.

```
//*--------------OBTAIN THE VALIDATION OBJECT AND GRAMMAR FUNCTION---*
//MAKEOBJ  EXEC SAS6
//SASLIB      DD DSN=TOOLKIT.LOAD,DISP=SHR
//MCBOBJ      DD DSN=TOOLKIT.SASC.OBJ(UWFONEOF),DISP=SHR
//CTRANS      DD DSN=SASC.LOAD,DISP=SHR
//            DD DSN=SASC.LINKLIB,DISP=SHR
//SYSIN       DD *
PROC USERPROC OBJECT=MCBOBJ MODNAME=UWFONEOF MODTYPE=FORMAT;
//*--------------COMPILE THE ITEM(S)---------------------------------*
//COMPILE  EXEC CCOMPILE,MEMBER=FMTC
//COMPILE  EXEC CCOMPILE,MEMBER=IFFEXTC
```

```
//*--------------CLINK THE ITEM(S)-----------------------------------*
//CLINK    EXEC CLINK
//SYSIN      DD *
INCLUDE COBJ(FMTC)              ALL C OBJECTS
INCLUDE COBJ(IFFEXTC)           ALL C OBJECTS
INCLUDE SYSLIB(L$UMAIN)         MAIN ROUTINE ENTRY NEEDED
INCLUDE SYSLIB(L$UEXIT)         EXIT ROUTINE ENTRY NEEDED
INCLUDE SYSLIB(L$UJUMP)         ENSURE BUGFIX OBJECT INCLUDED
//*--------------LINKEDIT THE IFF------------------------------------*
//LINKEDIT EXEC UWLINK
//SYSLIN      DD  *
 INCLUDE  MCBOBJ(UWFONEOF)      GENERATED BY PROC USERPROC
 INCLUDE  CLINKOBJ              ALL LOCAL CLINK-ED CODE
 INCLUDE  UWOBJ(IFFINT)         SUPPLIED BY SAS/TOOLKIT
 INCLUDE  UWOBJ(SASCFACE)       SUPPLIED BY SAS/TOOLKIT
 INCLUDE  UWOBJ(PRCINT)         FOR SAS_ CALLS
 ENTRY    IFFSASC               INDICATES SAS/C ENTRY
 NAME     UWFONEOF(R)
//
```

Compiling and Link-editing FORTRAN Programs

This section illustrates sample JCL procedures provided with SAS/TOOLKIT software to compile and link-edit procedures and IFFCs with the VS FORTRAN Compiler. Following the sample JCL procedures are examples of how to use these procedures for each of the following:

- procedures
- functions or CALL routines
- informats
- formats.

All of the JCL procedures and the JCL used to invoke these procedures use the following sample PDS names:

TOOLKIT.FTN.*type*
: are your libraries. The types can include source (SRC), object (OBJ), and load (LOAD) libraries.

TOOLKIT.GRM
: is your grammar source module.

TOOLKIT.*type*
: are the SAS/TOOLKIT libraries.

SITE.VSF2COMP
: is the FORTRAN load library at your site.

SITE.VSF2FORT
: is the FORTRAN subroutine library at your site.

Sample JCL Procedures for the FORTRAN Language

SAS/TOOLKIT software provides the following two sample JCL procedures:

UWFORT
: compiles a SAS module using the VS FORTRAN compiler.

UWLINKF
: runs the standard IBM linkage editor and links your SAS module with SAS/TOOLKIT software. If you are creating a SAS procedure, this step also links the grammar for your procedure with the main program.

You use these JCL procedures to compile and link-edit any SAS/TOOLKIT module written in the FORTRAN language. Your SAS Software Representative customizes these JCL procedures at the time that SAS/TOOLKIT software is installed at your site. Ask your SAS Software Representative to help you locate these JCL procedures if you need to examine them before using them.

Note that the JCL illustrated in this section assumes that your site has installed the UWFORT and UWLINKF procedures in a system procedure library at your site. If your site has not installed these procedures, ask your SAS Software Representative to provide you with the JCL included on the installation tape for SAS/TOOLKIT software. This JCL integrates the required procedures as instream procedures with the JCL illustrated in this section.

Note: A SAS/TOOLKIT load module can be loaded above the line with the MVS/XA operating system. This means that you should always use AMODE of 31 and RMODE of ANY. With those option choices, the module is loaded above the line if possible. If you are running under a non-XA system, the module is loaded below the line and everything functions normally. By default, the UWLINKF procedure specifies AMODE=31 and RMODE=ANY in the statement that runs the IEWL program.

FORTRAN Procedures

This JCL illustrates how to compile and link-edit a procedure with the FORTRAN VS Compiler.

Note: The SAS/TOOLKIT libraries for PL/I are required in the first step of this JCL.

```
//*--------------PREPROCESS THE FORTRAN CODE-------------------------*
//          EXEC SAS6
//IN          DD DSN=TOOLKIT.FTN.SRC(FUNCF),DISP=SHR
//OUT         DD DSN=&&TEMPSRC(FUNCF),DISP=(NEW,PASS),UNIT=SYSDA,
//               SPACE=(TRK,(10,10,10))
//PLIHFILE    DD DSN=TOOLKIT.PLI.MACLIB(UWHOST),DISP=SHR
//            DD DSN=TOOLKIT.PLI.MACLIB(UWIFFH),DISP=SHR
//SYSIN       DD DSN=TOOLKIT.TEST(FTNPREP),DISP=SHR
```

```
//*--------------OBTAIN THE VALIDATION OBJECT AND GRAMMAR FUNCTION---*
//MAKEOBJ  EXEC SAS6
//MCBOBJ     DD DSN=TOOLKIT.FTN.OBJ(EXAMPLE),DISP=SHR
//SYSIN      DD *
PROC USERPROC OBJECT=MCBOBJ MODNAME=EXAMPLE; RUN;
*-----CREATE THE GRAMMAR FUNCTION-----*;
PROC USERPROC GRAMMAR='TOOLKIT.GRM(EXAMPLE)' GRAMFUNC=EXMPLG
    FUNCFILE='TOOLKIT.FTN.SRC(EXMPLGF)'
    GRAMLANG=FORTRAN INCLLIB='TOOLKIT.GRM'; RUN;
//*--------------PREPROCESS THE FORTRAN CODE-------------------------*
//          EXEC SAS6
//SASLIB     DD DSN=TOOLKIT.LOAD,DISP=SHR
//IN         DD DSN=TOOLKIT.FTN.SRC(EXAMPLEF),DISP=SHR
//OUT        DD DSN=&&TEMPSRC(EXAMPLEF),DISP=(NEW,PASS),UNIT=SYSDA,
//             SPACE=(TRK,(10,10,10))
//PLIHFILE   DD DSN=TOOLKIT.PLI.MACLIB(VECDEFPI),DISP=SHR
//SYSIN      DD DSN=TOOLKIT.TEST(FTNPREP),DISP=SHR
//*--------------PREPROCESS THE FORTRAN CODE-------------------------*
//          EXEC SAS6
//SASLIB     DD DSN=TOOLKIT.LOAD,DISP=SHR
//IN         DD DSN=TOOLKIT.FTN.SRC(EXMPLGF),DISP=SHR
//OUT        DD DSN=&&TEMPSRC(EXMPLGF),DISP=(SHR,PASS)
//PLIHFILE   DD DSN=TOOLKIT.PLI.MACLIB(VECDEFPI),DISP=SHR
//SYSIN      DD DSN=TOOLKIT.TEST(FTNPREP),DISP=SHR
//*--------------COMPILE THE PROCEDURE-------------------------------*
//COMPILE  EXEC UWFORT,MEMBER=EXAMPLEF
//*--------------COMPILE THE GRAMMAR FUNCTION------------------------*
//COMPILE  EXEC UWFORT,MEMBER=EXMPLGF
//*--------------LINKEDIT THE FUNCTION-------------------------------*
//LINKEDIT EXEC UWLINKF
//SYSLIN     DD *
 INCLUDE  FTNOBJ(EXAMPLE)       GENERATED BY PROC USERPROC
 INCLUDE  FTNOBJ(EXAMPLEF)      USER-WRITTEN OBJECTS
 INCLUDE  FTNOBJ(EXMPLGF)       USER-WRITTEN OBJECTS
 INCLUDE  UWOBJ(PRCINT)         PROVIDED BY SAS/TOOLKIT
 INCLUDE  SYSLIB(EXIT)          FORTRAN EXIT ROUTINE
 ENTRY    ENTRY                 INDICATES PROCEDURE ENTRY
 NAME     EXAMPLE(R)            NAME OF MODULE
//
```

FORTRAN Functions and CALL Routines

This JCL illustrates how to compile and link-edit a function or CALL routine with the FORTRAN VS Compiler.

```
//*--------------OBTAIN THE VALIDATION OBJECT AND GRAMMAR FUNCTION---*
//MAKEOBJ  EXEC SAS6
//SASLIB     DD DSN=TOOLKIT.LOAD,DISP=SHR
//MCBOBJ     DD DSN=TOOLKIT.FTN.OBJ(UWUTESTF),DISP=SHR
//SYSIN      DD *
PROC USERPROC OBJECT=MCBOBJ MODNAME=UWUTESTF MODTYPE=FORMAT;
```

```
//*--------------COMPILE THE IFF-------------------------------------*
//COMPILE  EXEC UWFORT,MEMBER=FUNCF
//COMPILE  EXEC UWFORT,MEMBER=IFFEXTF,SRCLIB='TOOLKIT.FTN.SRC'
//*--------------LINKEDIT THE FUNCTION-------------------------------*
//LINKEDIT EXEC UWLINKF
//SYSLIN     DD *
 INCLUDE  FTNOBJ(UWUTESTF)      GENERATED BY PROC USERPROC
 INCLUDE  FTNOBJ(FUNCF)         USER-WRITTEN OBJECTS
 INCLUDE  FTNOBJ(IFFEXTF)       USER-WRITTEN OBJECTS
 INCLUDE  UWOBJ(IFFINT)         TOOLKIT-SUPPLIED OBJECTS
 INCLUDE  UWOBJ(FTNFACE)          "      "           "
 INCLUDE  UWOBJ(PRCINT)
 INCLUDE  SYSLIB(VFEIN#)        TO ENSURE FORTRAN CODE RESOLUTION
 ENTRY    IFFFTN                INDICATES FORTRAN IFF MODULE
 NAME     UWUTESTF(R)           NAME OF MODULE
//
```

FORTRAN Informats

This JCL illustrates how to compile and link-edit an informat with the VS FORTRAN Compiler.

```
//*--------------OBTAIN THE VALIDATION OBJECT AND GRAMMAR FUNCTION---*
//MAKEOBJ  EXEC SAS6
//SASLIB      DD DSN=TOOLKIT.LOAD,DISP=SHR
//MCBOBJ      DD DSN=TOOLKIT.FTN.OBJ(UWITESTI),DISP=SHR
//SYSIN       DD *
PROC USERPROC OBJECT=MCBOBJ MODNAME=UWITESTI MODTYPE=INFORMAT;
//*--------------COMPILE THE IFF-------------------------------------*
//COMPILE  EXEC UWFORT,MEMBER=INFMTF
//COMPILE  EXEC UWFORT,MEMBER=IFFEXTF,SRCLIB='TOOLKIT.FTN.SRC'
//*--------------LINKEDIT THE FUNCTION-------------------------------*
//LINKEDIT EXEC UWLINKF
//SYSLIN     DD *
 INCLUDE  FTNOBJ(UWITESTI)      GENERATED BY PROC USERPROC
 INCLUDE  FTNOBJ(INFMTF)        USER-WRITTEN OBJECTS
 INCLUDE  FTNOBJ(IFFEXTF)       USER-WRITTEN OBJECTS
 INCLUDE  UWOBJ(IFFINT)         TOOLKIT-SUPPLIED OBJECTS
 INCLUDE  UWOBJ(FTNFACE)          "      "           "
 INCLUDE  UWOBJ(PRCINT)
 INCLUDE  SYSLIB(VFEIN#)        TO ENSURE FORTRAN CODE RESOLUTION
 ENTRY    IFFFTN                INDICATES FORTRAN IFF MODULE
 NAME     UWITESTI(R)           NAME OF MODULE
//
```

FORTRAN Formats

This JCL illustrates how to compile and link-edit a format with the VS FORTRAN Compiler.

```
//*--------------OBTAIN THE VALIDATION OBJECT AND GRAMMAR FUNCTION---*
//MAKEOBJ  EXEC SAS6
//SASLIB     DD DSN=TOOLKIT.LOAD,DISP=SHR
//MCBOBJ     DD DSN=TOOLKIT.FTN.OBJ(UWFONEOF),DISP=SHR
//SYSIN      DD *
PROC USERPROC OBJECT=MCBOBJ MODNAME=UWFONEOF MODTYPE=FORMAT;
//*--------------COMPILE THE IFF------------------------------------*
//COMPILE  EXEC UWFORT,MEMBER=FMTF
//COMPILE  EXEC UWFORT,MEMBER=IFFEXTF,SRCLIB='TOOLKIT.FTN.SRC'
//*--------------LINKEDIT THE FUNCTION-------------------------------*
//LINKEDIT EXEC UWLINKF
//SYSLIN     DD *
 INCLUDE  FTNOBJ(UWFONEOF)      GENERATED BY PROC USERPROC
 INCLUDE  FTNOBJ(FMTF)          USER-WRITTEN OBJECTS
 INCLUDE  FTNOBJ(IFFEXTF)       USER-WRITTEN OBJECTS
 INCLUDE  UWOBJ(IFFINT)         TOOLKIT-SUPPLIED OBJECTS
 INCLUDE  UWOBJ(FTNFACE)          "      "          "
 INCLUDE  UWOBJ(PRCINT)
 INCLUDE  SYSLIB(VFEIN#)        TO ENSURE FORTRAN CODE RESOLUTION
 ENTRY    IFFFTN                INDICATES FORTRAN IFF MODULE
 NAME     UWFONEOF(R)           NAME OF MODULE
//
```

Compiling and Link-editing PL/I Programs

This section illustrates sample JCL procedures provided with SAS/TOOLKIT software to compile and link-edit procedures and IFFCs with the PL/I V2 Compiler. Following the sample JCL procedures are examples of how to use these procedures for each of the following:

- □ procedures
- □ functions or CALL routines
- □ informats
- □ formats.

All of the JCL procedures and the JCL used to invoke these procedures use the following sample PDS names:

TOOLKIT.PLI.*type*
: are your libraries. The types can include source (SRC), object (OBJ), and load (LOAD) libraries.

TOOLKIT.GRM
: is your grammar source module.

TOOLKIT.*type*
: are the SAS/TOOLKIT libraries.

SITE.PLICOMP
is the PL/I load library at your site.

SITE.PLIBASE
is the PL/I subroutine library at your site.

Sample JCL Procedures for the PL/I Language

SAS/TOOLKIT software provides the following two sample JCL procedures:

UWPLI
: compiles a SAS module using the PL/I V2 Compiler.

UWLINKP
: runs the standard IBM linkage editor and links your SAS module with SAS/TOOLKIT software. If you are creating a SAS procedure, this step also links the grammar for your procedure with the main program.

You use these JCL procedures to compile and link-edit any SAS/TOOLKIT module written in the PL/I language. Your SAS Software Representative customizes these JCL procedures at the time that SAS/TOOLKIT software is installed at your site. Ask your SAS Software Representative to help you locate these JCL procedures if you need to examine them before using them.

Note that the JCL illustrated in this section assumes that your site has installed the UWPLI and UWLINKP procedures in a system procedure library at your site. If your site has not installed these procedures, ask your SAS Software Representative to provide you with the JCL included on the installation tape for SAS/TOOLKIT software. This JCL integrates the required procedures as instream procedures with the JCL illustrated in this section.

Note: A SAS/TOOLKIT load module can be loaded above the line with the MVS/XA operating system. This means that you should always use AMODE of 31 and RMODE of ANY. With those option choices, the module is loaded above the line if possible. If you are running under a non-XA system, the module is loaded below the line and everything functions normally. By default, the UWLINKP procedure specifies AMODE=31 and RMODE=ANY in the statement that runs the IEWL program.

PL/I Procedures

This JCL illustrates how to compile and link-edit a procedure with the PL/I Compiler.

```
//*--------------OBTAIN THE VALIDATION OBJECT AND GRAMMAR FUNCTION---*
//MAKEOBJ  EXEC SAS6
//SASLIB     DD DSN=TOOLKIT.LOAD,DISP=SHR
//MCBOBJ     DD DSN=&&PLIOBJ(MCBOBJ),DISP=(NEW,PASS),UNIT=SYSDA,
//              SPACE=(TRK,(50,10,20)),
//              DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSIN      DD *
PROC USERPROC OBJECT=MCBOBJ MODNAME=EXAMPLE;
PROC USERPROC GRAMMAR='TOOLKIT.GRM(EXAMPLE)'
    GRAMLANG=PLI  INCLLIB='TOOLKIT.GRM'
```

```
    GRAMFUNC=EXMPLG FUNCFILE='TOOLKIT.PLI.SRC(EXMPLGP)';
//*--------------COMPILE THE ITEM(S)---------------------------------*
//COMPILE  EXEC UWPLI,MEMBER=EXAMPLEP
//COMPILE  EXEC UWPLI,MEMBER=EXMPLGP
//*--------------LINKEDIT THE PROCEDURE------------------------------*
//LINKEDIT EXEC UWLINKP
//SYSLIN      DD  *
 INCLUDE  SASOBJ(MCBOBJ)         GENERATED BY PROC USERPROC
 INCLUDE  SASOBJ(EXAMPLEP)       PL/I COMPILATION OF PROCEDURE
 INCLUDE  SASOBJ(EXMPLGP)        PL/I COMPILATION OF GRAMMAR FUNCTION
 INCLUDE  UWOBJ(PRCINT)          PROVIDED BY SAS/TOOLKIT
 ENTRY    ENTRY                  INDICATES PROCEDURE ENTRY
 NAME     EXAMPLE(R)             RESULTANT PROCEDURE MODULE NAME
```

PL/I Functions and CALL Routines

This JCL illustrates how to compile and link-edit a function or CALL routine with the PL/I Compiler.

```
//*--------------OBTAIN THE VALIDATION OBJECT AND GRAMMAR FUNCTION---*
//          EXEC SAS6
//SASLIB      DD DSN=TOOLKIT.LOAD,DISP=SHR
//MCBOBJ      DD DSN=TOOLKIT.PLI.OBJ(UWUTESTF),DISP=SHR
//SYSIN       DD *
*-----CREATE THE OBJECT-----*;
PROC USERPROC OBJECT=MCBOBJ MODNAME=UWUTESTF MODTYPE=FUNCTION;
//*--------------COMPILE THE IFF PIECES------------------------------*
//          EXEC UWPLI,MEMBER=FUNCP1
//          EXEC UWPLI,MEMBER=FUNCP2
//          EXEC UWPLI,MEMBER=FUNCP3
//          EXEC UWPLI,MEMBER=FUNCP4
//          EXEC UWPLI,MEMBER=FUNCP5
//          EXEC UWPLI,MEMBER=FUNCP6
//          EXEC UWPLI,MEMBER=IFFEXTP
//*--------------LINKEDIT THE FUNCTION-------------------------------*
//          EXEC UWLINKP
//UWLINKP.SYSLIN      DD  *
 INCLUDE  SASOBJ(UWUTESTF)       GENERATED BY PROC USERPROC
 INCLUDE  SASOBJ(FUNCP1)         USER-WRITTEN OBJECTS
 INCLUDE  SASOBJ(FUNCP2)           "     "        "
 INCLUDE  SASOBJ(FUNCP3)           "     "        "
 INCLUDE  SASOBJ(FUNCP4)           "     "        "
 INCLUDE  SASOBJ(FUNCP5)           "     "        "
 INCLUDE  SASOBJ(FUNCP6)           "     "        "
 INCLUDE  SASOBJ(IFFEXTP)          "     "        "
 INCLUDE  UWOBJ(IFFINT)          TOOLKIT-SUPPLIED OBJECTS
 INCLUDE  UWOBJ(PLIFACE)           "     "            "
 INCLUDE  UWOBJ(PRCINT)          FOR SAS_ CALLS
 ENTRY    IFFPLI                 INDICATES PL/I IFF MODULE
 NAME     UWUTESTF(R)            NAME OF MODULE
```

PL/I Informats

This JCL illustrates how to compile and link-edit an informat with the PL/I Compiler.

```
//*--------------OBTAIN THE VALIDATION OBJECT AND GRAMMAR FUNCTION---*
//         EXEC SAS6
//SASLIB     DD DSN=TOOLKIT.LOAD,DISP=SHR
//MCBOBJ     DD DSN=TOOLKIT.PLI.OBJ(UWITESTI),DISP=SHR
//SYSIN      DD *
*-----CREATE THE OBJECT-----*;
PROC USERPROC OBJECT=MCBOBJ MODNAME=UWITESTI MODTYPE=INFORMAT;
//*--------------COMPILE THE IFF PIECES------------------------------*
//         EXEC UWPLI,MEMBER=INFMTP1
//         EXEC UWPLI,MEMBER=INFMTP2
//         EXEC UWPLI,MEMBER=INFMTP3
//         EXEC UWPLI,MEMBER=IFFEXTP
//*--------------LINKEDIT THE FUNCTION-------------------------------*
//         EXEC UWLINKP
//UWLINKP.SYSLIN      DD  *
 INCLUDE  SASOBJ(UWITESTI)      GENERATED BY PROC USERPROC
 INCLUDE  SASOBJ(INFMTP1)       USER-WRITTEN OBJECTS
 INCLUDE  SASOBJ(INFMTP2)         "     "      "
 INCLUDE  SASOBJ(INFMTP3)         "     "      "
 INCLUDE  SASOBJ(IFFEXTP)         "     "      "
 INCLUDE  UWOBJ(IFFINT)         TOOLKIT-SUPPLIED OBJECTS
 INCLUDE  UWOBJ(PLIFACE)          "     "            "
 INCLUDE  UWOBJ(PRCINT)         FOR SAS_ CALLS
 ENTRY    IFFPLI                INDICATES PL/I IFF MODULE
 NAME     UWITESTI(R)           NAME OF MODULE
```

PL/I Formats

This JCL illustrates how to compile and link-edit a format with the PL/I Compiler.

```
//*--------------OBTAIN THE VALIDATION OBJECT AND GRAMMAR FUNCTION---*
//         EXEC SAS6
//SASLIB     DD DSN=TOOLKIT.LOAD,DISP=SHR
//MCBOBJ     DD DSN=TOOLKIT.PLI.OBJ(UWFONEOF),DISP=SHR
//SYSIN      DD *
*-----CREATE THE OBJECT-----*;
PROC USERPROC OBJECT=MCBOBJ MODNAME=UWFONEOF MODTYPE=FORMAT;
//*--------------COMPILE THE IFF PIECES------------------------------*
//         EXEC UWPLI,MEMBER=FMTP1
//         EXEC UWPLI,MEMBER=FMTP2
//         EXEC UWPLI,MEMBER=FMTP3
//         EXEC UWPLI,MEMBER=IFFEXTP
//*--------------LINKEDIT THE FUNCTION-------------------------------*
//         EXEC UWLINKP
//UWLINKP.SYSLIN      DD  *
 INCLUDE  SASOBJ(UWFONEOF)      GENERATED BY PROC USERPROC
 INCLUDE  SASOBJ(FMTP1)         USER-WRITTEN OBJECTS
```

```
INCLUDE  SASOBJ(FMTP2)          "    "    "
INCLUDE  SASOBJ(FMTP3)          "    "    "
INCLUDE  SASOBJ(IFFEXTP)        "    "    "
INCLUDE  UWOBJ(IFFINT)         TOOLKIT-SUPPLIED OBJECTS
INCLUDE  UWOBJ(PLIFACE)         "    "         "
INCLUDE  UWOBJ(PRCINT)         FOR SAS_ CALLS
ENTRY    IFFPLI                INDICATES PL/I IFF MODULE
NAME     UWFONEOF(R)           NAME OF MODULE
```

Assembling and Link-editing IBM 370 Assembler Programs

This section illustrates sample JCL procedures provided with SAS/TOOLKIT software to assemble and link-edit procedures and IFFCs with the IBM 370 assembler language. Following the sample JCL procedures are examples of how to use these procedures for each of the following:

- procedures
- functions or CALL routines
- informats
- formats.

All of the JCL procedures and the JCL used to invoke these procedures use the following sample PDS names:

TOOLKIT.ASM.*type*
: are your libraries. The types can include source (SRC), object (OBJ), and load (LOAD) libraries.

TOOLKIT.GRM
: is your grammar source module.

TOOLKIT.*type*
: are the SAS/TOOLKIT libraries.

Sample JCL Procedures for the IBM 370 Assembler Language

SAS/TOOLKIT software provides the following two sample JCL procedures:

UWASM
: assembles a SAS module using the IBM 370 assembler language.

UWLINKA
: runs the standard IBM linkage editor and links your SAS module with SAS/TOOLKIT software. If you are creating a SAS procedure, this step also links the grammar for your procedure with the main program.

You use these JCL procedures to assemble and link-edit any SAS/TOOLKIT module written in the IBM 370 assembler language. Your SAS Software Representative customizes these JCL procedures at the time that SAS/TOOLKIT software is installed at your site. Ask your SAS Software Representative to help you locate these JCL procedures if you need to examine them before using them.

Note that the JCL illustrated in this section assumes that your site has installed the UWASM and UWLINKA procedures in a system procedure library at your site. If your site has not installed these procedures, ask your SAS Software Representative to provide you with the JCL included on the installation tape for SAS/TOOLKIT software. This JCL integrates the required procedures as instream procedures with the JCL illustrated in this section.

Note: A SAS/TOOLKIT load module can be loaded above the line with the MVS/XA operating system. This means that you should always use AMODE of 31 and RMODE of ANY. With those option choices, the module is loaded above the line if possible. If you are running under a non-XA system, the module is loaded below the line and everything functions normally. By default, the UWLINKA procedure specifies AMODE=31 and RMODE=ANY in the statement that runs the IEWL program.

IBM 370 Assembler Procedures

This JCL illustrates how to assemble and link-edit a procedure with the IBM 370 Assembler.

```
//*--------------OBTAIN THE VALIDATION OBJECT AND GRAMMAR FUNCTION---*
//MAKEOBJ  EXEC SAS6
//SASLIB     DD DSN=TOOLKIT.LOAD,DISP=SHR
//MCBOBJ     DD DSN=TOOLKIT.ASM.OBJ(EXAMPLE),DISP=SHR
//SYSIN      DD *
PROC USERPROC OBJECT=MCBOBJ MODNAME=EXAMPLE;
PROC USERPROC GRAMMAR='TOOLKIT.GRM(EXAMPLE)' GRAMFUNC=EXMPLG
    FUNCFILE='TOOLKIT.ASM.SRC(EXMPLGA)'
    GRAMLANG=ASM370 INCLLIB='TOOLKIT.GRM'; RUN;
//*--------------ASSEMBLE THE PROCEDURE------------------------------*
//COMPILE  EXEC UWASM,MEMBER=EXAMPLEA
//*--------------ASSEMBLE THE GRAMMAR--------------------------------*
//COMPILE  EXEC UWASM,MEMBER=EXMPLGA
//*--------------LINKEDIT THE FUNCTION-------------------------------*
//LINKEDIT EXEC UWLINKF
//SYSLIN     DD *
 INCLUDE  ASMOBJ(EXAMPLE)       GENERATED BY PROC USERPROC
 INCLUDE  ASMOBJ(EXAMPLEA)      USER-WRITTEN OBJECT
 INCLUDE  ASMOBJ(EXMPLGA)       ASM GRAMMAR FUNCTION
 INCLUDE  UWOBJ(PRCINT)         SUPPLIED BY SAS/TOOLKIT
 ENTRY    ENTRY                 INDICATES PROCEDURE
 NAME     EXAMPLE(R)            NAME OF MODULE
//
```

IBM 370 Assembler Functions and CALL Routines

This JCL illustrates how to assemble and link-edit a function or CALL routine with the IBM 370 Assembler.

```
//*--------------OBTAIN THE VALIDATION OBJECT AND GRAMMAR FUNCTION---*
//MAKEOBJ  EXEC SAS6
//SASLIB     DD DSN=TOOLKIT.LOAD,DISP=SHR
//MCBOBJ     DD DSN=TOOLKIT.ASM.OBJ(UWUTESTF),DISP=SHR
//SYSIN      DD *
PROC USERPROC OBJECT=MCBOBJ MODNAME=UWUTESTF MODTYPE=FUNCTION;
//*--------------ASSEMBLE THE IFF-----------------------------------*
//COMPILE  EXEC UWASM,MEMBER=FUNCA
//*--------------LINKEDIT THE FUNCTION------------------------------*
//LINKEDIT EXEC UWLINKF
//SYSLIN     DD *
 INCLUDE  ASMOBJ(UWUTESTF)       GENERATED BY PROC USERPROC
 INCLUDE  ASMOBJ(FUNCA)          USER-WRITTEN OBJECTS
 INCLUDE  UWOBJ(IFFINT)          TOOLKIT-SUPPLIED OBJECTS
 INCLUDE  UWOBJ(ASMFACE)           "      "         "
 ENTRY    IFFASM                 INDICATES FORTRAN IFF MODULE
 NAME     UWUTESTF(R)            NAME OF MODULE
//
```

IBM 370 Assembler Informats

This JCL illustrates how to assemble and link-edit an informat with the IBM 370 Assembler.

```
//*--------------OBTAIN THE VALIDATION OBJECT AND GRAMMAR FUNCTION---*
//MAKEOBJ  EXEC SAS6
//SASLIB     DD DSN=TOOLKIT.LOAD,DISP=SHR
//MCBOBJ     DD DSN=TOOLKIT.ASM.OBJ(UWITESTI),DISP=SHR
//SYSIN      DD *
PROC USERPROC OBJECT=MCBOBJ MODNAME=UWITESTI MODTYPE=INFORMAT;
//*--------------ASSEMBLE THE IFF-----------------------------------*
//COMPILE  EXEC UWASM,MEMBER=INFMTA
//*--------------LINKEDIT THE FUNCTION------------------------------*
//LINKEDIT EXEC UWLINKF
//SYSLIN     DD *
 INCLUDE  ASMOBJ(UWITESTI)       GENERATED BY PROC USERPROC
 INCLUDE  ASMOBJ(INFMTA)         USER-WRITTEN OBJECTS
 INCLUDE  UWOBJ(IFFINT)          TOOLKIT-SUPPLIED OBJECTS
 INCLUDE  UWOBJ(ASMFACE)           "      "         "
 INCLUDE  UWOBJ(PRCINT)          FOR SAS_ CALLS
 ENTRY    IFFASM                 INDICATES FORTRAN IFF MODULE
 NAME     UWITESTI(R)            NAME OF MODULE
//
```

IBM 370 Assembler Formats

This JCL illustrates how to assemble and link-edit a format with the IBM 370 Assembler.

```
//*--------------OBTAIN THE VALIDATION OBJECT AND GRAMMAR FUNCTION---*
//MAKEOBJ  EXEC SAS6
//SASLIB     DD DSN=TOOLKIT.LOAD,DISP=SHR
//MCBOBJ     DD DSN=TOOLKIT.ASM.OBJ(UWFONEOF),DISP=SHR
//SYSIN      DD *
PROC USERPROC OBJECT=MCBOBJ MODNAME=UWFONEOF MODTYPE=FORMAT;
//*--------------ASSEMBLE THE IFF------------------------------------*
//COMPILE  EXEC UWASM,MEMBER=FMTA
//*--------------LINKEDIT THE FUNCTION-------------------------------*
//LINKEDIT EXEC UWLINKF
//SYSLIN     DD *
 INCLUDE  ASMOBJ(UWFONEOF)      GENERATED BY PROC USERPROC
 INCLUDE  ASMOBJ(FMTA)          USER-WRITTEN OBJECTS
 INCLUDE  UWOBJ(IFFINT)         TOOLKIT-SUPPLIED OBJECTS
 INCLUDE  UWOBJ(ASMFACE)          "     "          "
 INCLUDE  UWOBJ(PRCINT)         FOR SAS_ CALLS
 ENTRY    IFFASM                INDICATES FORTRAN IFF MODULE
 NAME     UWFONEOF(R)           NAME OF MODULE
//
```

Appendix 3 Using SAS/TOOLKIT™ Software under VMS™

Introduction

This appendix provides sample .COM files for compiling and linking your procedure, function, CALL routine, informat, or format in the various supported languages. Under VMS, you can create procedures and IFFCs using the VAX C Compiler, the VAX PL/I Compiler, or the VAX FORTRAN Compiler. In addition, this appendix discusses how to use user-written SAS modules after they are compiled and how to make those modules available to other sites.

Accessing User-Written SAS Modules

This section discusses how to access user-written modules and how to make these modules available to other sites.

Specifying Where to Find Modules

Procedures and IFFCs are executable modules that are stored as .EXE files.

After you create your executable modules, you (and your users) can access them in a variety of ways. One easy way is to define a logical word that matches the executable name using the DEFINE command:

```
$ DEFINE ABCDEF [MYLIB]ABCDEF.EXE
```

This is the method used in the sample .COM files to access the SASUSERP (PROC USERPROC) executable module. Note that the current directory is always searched for executable modules.

An alternate method you may want to use when you are developing your program is to redefine the SAS$EXTENSION logical name to refer to the directory where your executable modules are stored. The directory referred to by this symbol is automatically checked when searching for executable modules. When you complete a SAS user-written module, you may want to have the SAS Software Representative copy the executable modules into a public directory which can be accessed by all users.

Distributing Modules to Other SAS Sites

If you distribute a module you have written to other SAS sites, you may also need to make available the SASPROCI module that is supplied with SAS/TOOLKIT software. You are free to redistribute SASPROCI since it is treated as a transient module. If the site to which you are distributing your SAS module has Release 6.07 or later of the SAS System, you probably do not need to supply the SASPROCI module. If you do not supply SASPROCI and the site receiving your module does not already have it, your procedure or IFFC will not run at that site.

Note: You are *not* permitted to redistribute any other components of SAS/TOOLKIT software.

Symbol Definitions

To simplify the use of SAS/TOOLKIT on VMS, the following examples use a group of symbols that correspond to the different sample directories provided with SAS/TOOLKIT software. Table A3.1 lists the symbols and describes the directories to which they refer.

Table A3.1 *Symbols for VMS Directories*

Symbol	Description of Directory Referenced by Symbol
TOOLKIT_OBJ	language-independent objects
TOOLKIT_C_OBJ	sample C code compiled
TOOLKIT_PLI_OBJ	sample PL/I code compiled

(*continued*)

Table A3.1
(continued)

Symbol	Description of Directory Referenced by Symbol
TOOLKIT_FTN_OBJ	sample FORTRAN code compiled
TOOLKIT_C_LOAD	sample executable modules from linking C code
TOOLKIT_PLI_LOAD	sample executable modules from linking PL/I code
TOOLKIT_FTN_LOAD	sample executable modules from linking FORTRAN code
TOOLKIT_LOAD	USERPROC and SASPROCI executable modules
TOOLKIT_TEST	SAS programs for testing and preprocessing

Creating Executable Modules

Executable modules are created by using the VMS LINK command after compiling the different components. One of the components is a program constants object that is created by PROC USERPROC. All the other components are created by compiling with the compiler of choice. The installation tape comes with several sample .COM files that you can use as a model in building .COM files for your own linking command files.

Understanding the Sample .COM Files

This section describes the general structure of the .COM files provided with SAS/TOOLKIT software. Later sections in this chapter list the sample .COM file for each type of user-written module:

- □ procedures
- □ functions or CALL routines
- □ informats
- □ formats.

Note: Unlike MVS or CMS, the VMS compilers are all very compatible, so that linking compiled objects is the same regardless of the compiler used.

The following portion of a .COM file shows how to create the program constants object file using PROC USERPROC. Creating the program constants object file is a required step for all user-written modules.

This part of the sample creates a temporary SAS program called TEMP.SAS using the CREATE command. The CREATE command allows you to have statements in-line in the .COM file to be copied to TEMP.SAS. All lines up to, but not including, the line beginning with the $ (dollar sign) symbol are copied to TEMP.SAS. Once TEMP.SAS is created, the example runs the TEMP.SAS program with Version 6 of the SAS System so that PROC USERPROC can build the object file UWUTESTF.OBJ in the TOOLKIT_C_OBJ directory.

```
$ ! IFFJOBUC: .COM file to compile and link the sample
$ ! functions in C.
$ !
$ ! First create the pgm constants object with USERPROC
$ !
$ create temp.sas
  proc userproc object='toolkit_c_obj:uwutestf.obj'
                modtype=function modname=uwutestf;
$ define/nolog userproc toolkit_load:sasuserp.exe
$ sas606 temp
$ del           noconfirm temp.*;*
```

Note: When you are writing a SAS procedure, this step of the .COM file also includes a PROC USERPROC step to create the grammar function.

This portion of the .COM file compiles a C function named **funcc.c**, as well as the **iffextc** module from the SAS/TOOLKIT source library. User-written IFFCs must always have an IFFEXT routine. You can either code this routine in your source module, or if you don't have any special exit processing, include the module in the linkage steps, as is done here. (Refer to the discussion of the IFFEXT routine in Chapters 13, 14, or 15, depending on what kind of IFFC you are writing.)

```
$ !
$ ! now compile all components
$ @ccompile funcc
$ @ccompile iffextc
$ !
```

After all the necessary components are compiled, you can link the .EXE file. Always specify the SHARE option when linking since user-written modules must be shareable images. The list of objects includes all those compiled earlier, plus the program constants object created by PROC USERPROC. The last item being linked is an OPTIONS file called EXTRA.OPT, which is illustrated after this step.

```
$ ! now link all components
$ link/share/exe=toolkit_c_load:uwutestf.exe -
 toolkit_c_obj:funcc.obj,-
 toolkit_c_obj:iffextc.obj,-
 toolkit_c_obj:uwutestf.obj,-
 toolkit_c_cntl:extra/opt
$ exit
```

The EXTRA.OPT file contains the following:

```
TOOLKIT_OBJ:TOOLKIT/LIB
SASSHR/SHARE
GSMATCH=EQUAL,1,1
```

The TOOLKIT.OLB file referenced in the EXTRA.OPT file is the object library that resolves the 'SAS_' external references in your procedure or IFFC and the FNC or FMT references in IFFCs.

Compiling and Linking C Programs

This section illustrates the sample .COM files for compiling and linking procedures and IFFCs written in C. The first .COM file in the following sections shows the @CCOMPILE command file, which is used by the rest of the .COM files to perform the compilation.

The CCOMPILE Command File

You use the CCOMPILE .COM file to compile any SAS/TOOLKIT module written in the C language. Your SAS Software Representative customizes the CCOMPILE .COM file when SAS/TOOLKIT software is installed at your site. Ask your SAS Software Representative to help you locate this sample .COM file. You should print a copy of this file and look at it before using the other .COM files described in the following sections for C programs.

C Procedures

This .COM file illustrates the steps needed to compile and link a procedure written in the C language. The general structure of the .COM file is discussed earlier in the chapter in "Understanding the Sample .COM Files." The following .COM file is slightly different from the sample file discussed earlier because for procedures you must also link the grammar function, produced by PROC USERPROC. For procedures, the step in the .COM file that creates the PROC USERPROC statements also includes statements to generate the grammar function.

Listing A3.1
PRCJOBC2 .COM File

```
$ ! PRCJOBC2: COM file to compile and link the sample
$ ! procedure EXAMPLE in C.
$ !
$ ! First create the pgm constants object and the
$ ! grammar function source.
$ create temp.sas
     proc userproc grammar='toolkit_grm:example.grm'
                   gramlang=c gramfunc=exmplg
                   incllib='toolkit_grm:'
                   funcfile='[-.src]exmplgc.c'
                   noprint;
     proc userproc object='toolkit_c_obj:sasexamp.obj'
                   modtype=proc modname=sasexamp;
$ define/nolog sasuserp toolkit_load:sasuserp.exe
$ sas606 temp
$ del/noconfirm temp.*;*
$ !
$ ! Now compile all components.
$ @ccompile examplec
$ @ccompile exmplgc
$ !
$ ! Now link all components.
$ link/share/exe=toolkit_c_load:sasexamp.exe -
 toolkit_c_obj:examplec.obj,-
 toolkit_c_obj:exmplgc.obj,-
 toolkit_c_obj:sasexamp.obj,-
 toolkit_c_cntl:extra/opt
$ exit
```

C Functions or CALL Routines

This .COM file illustrates the steps needed to compile and link a function or CALL routine written in the C language. The general structure of the .COM file is discussed earlier in the chapter in "Understanding the Sample .COM Files."

Listing A3.2
IFFJOBUC .COM File

```
$ ! IFFJOBUC: COM file to compile and link the sample
$ ! functions in C.
$ !
$ ! First create the pgm constants object with USERPROC
$ !
$ create temp.sas
  proc userproc object='toolkit_c_obj:uwutestf.obj'
                modtype=function modname=uwutestf;
$ define/nolog userproc toolkit_load:sasuserp.exe
$ sas606 temp
$ del/noconfirm temp.*;*
$ !
$ ! now compile all components
$ @compile funcc
$ @compile iffextc
$ !
$ ! now link all components
$ link/share/exe=toolkit_c_load:uwutestf.exe -
 toolkit_c_obj:funcc.obj,-
 toolkit_c_obj:iffextc.obj,-
 toolkit_c_obj:uwutestf.obj,-
 toolkit_c_cntl:extra/opt
$ exit
```

C Informats

This .COM file illustrates the steps needed to compile and link an informat written in the C language. The general structure of the .COM file is discussed earlier in the chapter in "Understanding the Sample .COM Files."

Listing A3.3
IFFJOBIC .COM File

```
$ ! IFFJOBIC: COM file to compile and link the sample
$ ! informats in C.
$ !
$ ! First create the pgm constants object with USERPROC
$ !
$ create temp.sas
  proc userproc object='toolkit_c_obj:uwitesti.obj'
                modtype=informat modname=uwitesti;
$ define/nolog userproc toolkit_load:sasuserp.exe
$ sas606 temp
$ del/noconfirm temp.*;*
$ !
$ ! now compile all components
$ @compile infmtc
$ @compile iffextc
$ !
$ ! now link all components
$ link/share/exe=toolkit_c_load:uwitesti.exe -
 toolkit_c_obj:infmtc.obj,-
 toolkit_c_obj:iffextc.obj,-
 toolkit_c_obj:uwitesti.obj,-
 toolkit_c_cntl:extra/opt
$ exit
```

C Formats

This .COM file illustrates the steps needed to compile and link a format written in the C language. The general structure of the .COM file is discussed earlier in the chapter in "Understanding the Sample .COM Files."

Listing A3.4
IFFJOBFC .COM File

```
$ ! IFFJOBFC: COM file to compile and link the sample
$ ! formats in C.
$ !
$ ! First create the pgm constants object with USERPROC
$ !
$ create temp.sas
  proc userproc object='toolkit_c_obj:uwfoneof.obj'
               modtype=format modname=uwfoneof;
$ define/nolog userproc toolkit_load:sasuserp.exe
$ sas606 temp
$ del/noconfirm temp.*;*
$ !
$ ! now compile all components
$ @ccompile funcc
$ @ccompile iffextc
$ !
$ ! now link all components
$ link/share/exe=toolkit_c_load:uwfoneof.exe -
 toolkit_c_obj:fmtc.obj,-
 toolkit_c_obj:iffextc.obj,-
 toolkit_c_obj:uwfoneof.obj,-
 toolkit_c_cntl:extra/opt
$ exit
```

Compiling and Linking PL/I Programs

This section illustrates the sample .COM files for compiling and linking procedures and IFFCs written in the PL/I language. The first .COM file in the following sections shows the @UWPLI command file, which is used by the rest of the .COM files to perform the compilation.

The UWPLI Command File

You use the UWPLI .COM file to compile any SAS/TOOLKIT module written in the PL/I language. Your SAS Software Representative customizes the UWPLI .COM file when SAS/TOOLKIT software is installed at your site. Ask your SAS Software Representative to help you locate this sample .COM file. You should print a copy of this file and look at it before using the other .COM files described in the following sections for PL/I programs.

PL/I Procedures

This .COM file illustrates the steps needed to compile and link a procedure written in the PL/I language. The general structure of the .COM file is discussed earlier in the chapter in "Understanding the Sample .COM Files." The following .COM file is slightly different from the sample file discussed earlier because for procedures you must also link the grammar function, produced by PROC USERPROC. For procedures, the step in the .COM file that creates the PROC USERPROC statements also includes statements to generate the grammar function.

Listing A3.5
PRCJOBP2 .COM File

```
$ ! PRCJOBP2: COM file to compile and link the sample
$ ! procedure EXAMPLE in PL/I.
$ !
$ ! First create the pgm constants object and the
$ ! grammar function source.
$ create temp.sas
     proc userproc grammar='toolkit_grm:example.grm'
                   gramlang=pli gramfunc=exmplg
                   incllib='toolkit_grm:'
                   funcfile='[-.src]exmplgp.pli'
                   noprint;
     proc userproc object='toolkit_pli_obj:sasexamp.obj'
                   modtype=proc modname=sasexamp;
$ define/nolog sasuserp toolkit_load:sasuserp.exe
$ sas606 temp
$ del/noconfirm temp.*;*
$ !
$ ! Now compile all components.
$ @uwpli examplep
$ @uwpli exmplgp
$ !
$ ! Now link all components.
$ link/share/exe=toolkit_pli_load:sasexamp.exe -
 toolkit_pli_obj:examplep.obj,-
 toolkit_pli_obj:exmplgp.obj,-
 toolkit_pli_obj:sasexamp.obj,-
 toolkit_pli_cntl:extra/opt
$ exit
```

PL/I Functions or CALL Routines

This .COM file illustrates the steps needed to compile and link a function or CALL routine written in the PL/I language. The general structure of the .COM file is discussed earlier in the chapter in "Understanding the Sample .COM Files."

Listing A3.6
IFFJOBUP .COM File

```
$ ! IFFJOBUP: COM file to compile and link the sample
$ ! functions in PL/I.
$ !
$ ! First create the pgm constants object with USERPROC
$ !
$ create temp.sas
  proc userproc object='toolkit_pli_obj:uwutestf.obj'
                modtype=function modname=uwutestf;
$ define/nolog userproc toolkit_load:sasuserp.exe
$ sas606 temp
$ del/noconfirm temp.*;*
$ !
$ ! now compile all components
$ @uwpli funcp1
$ @uwpli funcp2
$ @uwpli funcp3
$ @uwpli funcp4
$ @uwpli funcp5
$ @uwpli funcp6
$ @uwpli iffextp
$ ! now link all components
$ link/share/exe=toolkit_pli_load:uwutestf.exe -
 toolkit_pli_obj:funcp1.obj,-
 toolkit_pli_obj:funcp2.obj,-
 toolkit_pli_obj:funcp3.obj,-
 toolkit_pli_obj:funcp4.obj,-
 toolkit_pli_obj:funcp5.obj,-
 toolkit_pli_obj:funcp6.obj,-
 toolkit_pli_obj:iffextp.obj,-
 toolkit_pli_obj:uwutestf.obj,-
 toolkit_pli_cntl:extra/opt
$ exit
```

PL/I Informats

This .COM file illustrates the steps needed to compile and link an informat written in the PL/I language. The general structure of the .COM file is discussed earlier in the chapter in "Understanding the Sample .COM Files."

Listing A3.7
IFFJOBIP .COM File

```
$ ! IFFJOBIP: COM file to compile and link the sample
$ ! informats in PL/I.
$ !
$ ! First create the pgm constants object with USERPROC
$ !
$ create temp.sas
  proc userproc object='toolkit_pli_obj:uwitesti.obj'
                modtype=informat modname=uwitesti;
$ define/nolog userproc toolkit_load:sasuserp.exe
$ sas606 temp
$ del/noconfirm temp.*;*
$ !
$ ! now compile all components
$ @uwpli infmtp1
$ @uwpli infmtp2
$ @uwpli infmtp3
$ @uwpli iffextp
$ ! now link all components
$ link/share/exe=toolkit_pli_load:uwitesti.exe -
 toolkit_pli_obj:infmtp1.obj,-
 toolkit_pli_obj:infmtp2.obj,-
 toolkit_pli_obj:infmtp3.obj,-
 toolkit_pli_obj:iffextp.obj,-
 toolkit_pli_obj:uwitesti.obj,-
 toolkit_pli_cntl:extra/opt
$ exit
```

PL/I Formats

This .COM file illustrates the steps needed to compile and link a format written in the PL/I language. The general structure of the .COM file is discussed earlier in the chapter in "Understanding the Sample .COM Files."

Listing A3.8
IFFJOBFP .COM File

```
$ ! IFFJOBFP: COM file to compile and link the sample
$ ! formats in PL/I.
$ !
$ ! First create the pgm constants object with USERPROC
$ !
$ create temp.sas
  proc userproc object='toolkit_pli_obj:uwfoneof.obj'
                modtype=format modname=uwfoneof;
$ define/nolog userproc toolkit_load:sasuserp.exe
$ sas606 temp
$ del/noconfirm temp.*;*
$ !
$ ! now compile all components
$ @uwpli fmtp1
$ @uwpli fmtp2
$ @uwpli fmtp3
$ @uwpli iffextp
$ ! now link all components
$ link/share/exe=toolkit_pli_load:uwfoneof.exe -
 toolkit_pli_obj:fmtp1.obj,-
 toolkit_pli_obj:fmtp2.obj,-
 toolkit_pli_obj:fmtp3.obj,-
 toolkit_pli_obj:iffextp.obj,-
 toolkit_pli_obj:uwfoneof.obj,-
 toolkit_pli_cntl:extra/opt
$ exit
```

Compiling and Linking FORTRAN Programs

This section illustrates the sample .COM files for compiling and linking procedures and IFFCs written in the FORTRAN language. The first .COM file in the following sections shows the @UWFORT command file, which is used by the rest of the .COM files to perform the compilation.

The UWFORT Command File

You use the UWFORT .COM file to compile any SAS/TOOLKIT module written in the FORTRAN language. Your SAS Software Representative customizes the UWFORT .COM file when SAS/TOOLKIT software is installed at your site. Ask your SAS Software Representative to help you locate this sample .COM file. You should print a copy of this file and look at it before using the other .COM files described in the following sections for FORTRAN programs.

FORTRAN Procedures

This .COM file illustrates the steps needed to compile and link a procedure written in the FORTRAN language. The general structure of the .COM file is discussed earlier in the chapter in "Understanding the Sample .COM Files." The following .COM file is slightly different from the sample file discussed earlier because for procedures you must also link the grammar function, produced by PROC USERPROC. For procedures, the step in the .COM file that creates the PROC USERPROC statements also includes statements to generate the grammar function.

Note that the MODIMAGE command must be used with FORTRAN procedures. The MODIMAGE commands (MODIMAGE.EXE and MODIMAGE.CLD) are included in the SAS/TOOLKIT load library. MODIMAGE modifies the characteristics of the image so that it is not tagged as writable or shareable.

Listing A3.9
PRCJOBF2 .COM File

```
$ ! PRCJOBF2: COM file to compile and link the sample
$ ! procedure EXAMPLE in FORTRAN.
$ !
$ ! First create the pgm constants object and the
$ ! grammar function source.
$ create temp.sas
     proc userproc grammar='toolkit_grm:example.grm'
                   gramlang=fortran gramfunc=exmplg
                   incllib='toolkit_grm:'
                   funcfile='[-.src]exmplgf.for'
                   noprint;
     proc userproc object='toolkit_ftn_obj:sasexamp.obj'
                   modtype=proc modname=sasexamp;
$ define/nolog sasuserp toolkit_load:sasuserp.exe
$ sas606 temp
$ del/noconfirm temp.*;*
$ !
$ ! Now compile all components.
$ @uwfort examplef
$ @uwfort exmplgf
$ !
$ ! Now link all components.
$ link/debug/share/exe=toolkit_ftn_load:sasexamp.exe -
 toolkit_ftn_obj:examplef.obj,-
 toolkit_ftn_obj:exmplgf.obj,-
 toolkit_ftn_obj:sasexamp.obj,-
 toolkit_ftn_cntl:extra/opt
$ set command disk:[yourdir]modimage.cld
$ modimage/shr/log toolkit_ftn_load:sasexamp.exe
$ exit
```

FORTRAN Functions or CALL Routines

This .COM file illustrates the steps needed to compile and link a function or CALL routine written in the FORTRAN language. The general structure of the .COM file is discussed earlier in the chapter in "Understanding the Sample .COM Files."

Listing A3.10
IFFJOBUF .COM File

```
$ ! IFFJOBUF: COM file to compile and link the sample
$ ! functions in FORTRAN.
$ !
$ ! First create the pgm constants object with USERPROC
$ !
$ create temp.sas
  proc userproc object='toolkit_ftn_obj:uwutestf.obj'
                modtype=function modname=uwutestf;
$ define/nolog userproc toolkit_load:sasuserp.exe
$ sas606 temp
$ del/noconfirm temp.*;*
$ !
$ ! now compile all components
$ @uwfort funcf
$ @uwfort iffextf
$ !
$ ! now link all components
$ link/share/exe=toolkit_ftn_load:uwutestf.exe -
 toolkit_ftn_obj:funcf.obj,-
 toolkit_ftn_obj:iffextf.obj,-
 toolkit_ftn_obj:uwutestf.obj,-
 toolkit_ftn_cntl:extra/opt
$ set command disk:[yourdir]modimage.cld
$ modimage/shr/log toolkit_ftn_load:uwutestf.exe
$ exit
```

FORTRAN Informats

This .COM file illustrates the steps needed to compile and link an informat written in the FORTRAN language. The general structure of the .COM file is discussed earlier in the chapter in "Understanding the Sample .COM Files."

Listing A3.11
IFFJOBIF .COM File

```
$ ! IFFJOBIF: COM file to compile and link the sample
$ ! informats in FORTRAN.
$ !
$ ! First create the pgm constants object with USERPROC
$ !
$ create temp.sas
  proc userproc object='toolkit_ftn_obj:uwitesti.obj'
                modtype=informat modname=uwitesti;
$ define/nolog userproc toolkit_load:sasuserp.exe
$ sas606 temp
$ del/noconfirm temp.*;*
$ !
$ ! now compile all components
$ @uwfort infmtf
$ @uwfort iffextf
$ !
$ ! now link all components
$ link/share/exe=toolkit_ftn_load:uwitesti.exe -
 toolkit_ftn_obj:infmtf.obj,-
 toolkit_ftn_obj:iffextf.obj,-
 toolkit_ftn_obj:uwitesti.obj,-
 toolkit_ftn_cntl:extra/opt
$ set command disk:[yourdir]modimage.cld
$ modimage/shr/log toolkit_ftn_load:uwitesti.exe
$ exit
```

FORTRAN Formats

This .COM file illustrates the steps needed to compile and link a format written in the FORTRAN language. The general structure of the .COM file is discussed earlier in the chapter in "Understanding the Sample .COM Files."

Listing A3.12
IFFJOBFF .COM File

```
$ ! IFFJOBFF: COM file to compile and link the sample
$ ! formats in FORTRAN.
$ !
$ ! First create the pgm constants object with USERPROC
$ !
$ create temp.sas
  proc userproc object='toolkit_ftn_obj:uwfoneof.obj'
                modtype=format modname=uwfoneof;
$ define/nolog userproc toolkit_load:sasuserp.exe
$ sas606 temp
$ del/noconfirm temp.*;*
$ !
$ ! now compile all components
$ @uwfort fmtf
$ @uwfort iffextf
$ ! now link all components
$ link/share/exe=toolkit_ftn_load:uwfoneof.exe -
 toolkit_ftn_obj:fmtf.obj,-
 toolkit_ftn_obj:iffextf.obj,-
 toolkit_ftn_obj:uwfoneof.obj,-
 toolkit_ftn_cntl:extra/opt
$ set command disk:[yourdir]modimage.cld
$ modimage/shr/log toolkit_ftn_load:uwfoneof.exe
$ exit
```

Glossary

aggregate storage location
a group of distinct files on an operating system. Different host operating systems call an aggregate grouping of files different names, such as a directory, a maclib, or a partitioned data set.

alternation
a group of terminals, lexicals, or production names separated by vertical bars. The vertical bars indicate that any one of the items can occur at a time.

argument
any word that follows the keyword in a SAS statement. In SAS/TOOLKIT software, SAS arguments are divided into two groups: bit arguments and parameter arguments.

base array
an array used as an anchor point in memory. The location of other data in memory can be expressed as an index to the base array.

bit argument
a SAS argument that can be turned on or off.

BY group
all observations with the same values for all BY variables.

BY-group processing
the process of using the BY statement to process observations that are ordered, grouped, or indexed according to the values of one or more variables. Many SAS procedures and the DATA step support BY-group processing. For example, you can use BY-group processing with the PRINT procedure to print separate reports for different groups of observations in a single SAS data set.

BY value
the value of a BY variable.

BY variable
a variable that is identified in a BY statement and whose values define groups of observations to be processed.

call-by-address language
a programming language that passes the address of a function's arguments when a function is invoked. When using this type of language, you can change the value of an argument within the function. Both PL/I and FORTRAN are call-by-address languages.

call-by-value language
a programming language that passes the actual values of a function's arguments when a function is invoked. When using this type of language, you cannot directly change the value of an argument within the function. The C language is a

call-by-value language. In addition, the implementation of SAS/TOOLKIT software for IBM 370 assembler language treats that language as call-by-value.

CALL routine
a program that can be called in a DATA step or in SCL programs by issuing a CALL statement. A CALL routine may change the value of some of the arguments passed to it, but it does not return a value as a function does. Some CALL routines are included with the SAS System; you can write others with SAS/TOOLKIT software. See also user-written CALL routine.

character descriptor
See dope vector.

character format
instructions to the SAS System to write character data values using a specific pattern.

character function
a function that enables you to perform character string manipulations.

character informat
instructions to the SAS System to read character data values into character variables using a specific pattern.

compilation
the process of checking syntax and translating a portion of a program into a form that the computer can execute.

data set reference
a SAS argument that specifies a SAS data set similar to DATA=*libref.member* or OUT=*libref.member*.

data set variable list
the list of all variables in a SAS data set. When the data set is opened, this list is created internally. All other lists of variables are actually lists of integers that represent the variable's position in the data set variable list. See also variable list.

data structure
a group of related data items stored in memory. In FORTRAN, data structures are called COMMON blocks; in IBM 370 assembler, data structures are called DSECTs. C and PL/I use the term data structures.

dope vector
a description of a character variable that is built internally when a character value is passed to a function or subroutine. The dope vector usually contains the length and the address of the character data.

double
a variable type that contains floating-point, double-precision data.

engine
a part of the SAS System that reads from or writes to a file. Each engine allows the SAS System to access files with a particular format. There are several types of engines.

executable module
a file of compiled and linked code. This is a machine-readable file that can be executed by the host operating system on which it was compiled and linked. See also object module and source module.

external file
a file maintained by the host operating system rather than by the SAS System. External files can contain raw data, SAS programming statements, procedure output, or output created by the PUT statement. See also fileref.

fileid pointer
a pointer returned when a SAS data set or SAS utility file is opened that provides a reference for any SAS_X routine accessing the file.

fileref
the name used to reference and identify an external file to the SAS System. You assign a fileref with a FILENAME statement or with host-specific commands.

Do not confuse filerefs with librefs. Filerefs are used for external files; librefs are used for SAS data libraries. See also libref.

format
an instruction the SAS System uses to display or write each value of a variable. There are two types of formats: formats supplied by SAS software and user-written formats created using the FORMAT procedure in base SAS software or using SAS/TOOLKIT software. See also user-written format.

function
in base SAS software, a routine that can accept arguments, perform an operation, and return a value. For example, the ABS function returns the absolute value of a numeric argument. Functions can return either numeric or character results from numeric or character arguments, respectively. Some functions are included with the SAS System; you can write others with SAS/TOOLKIT software.

gather writing
the process of writing an observation to a SAS data set. The separate variables defined in your program are gathered into the output buffer and then written as a single observation to the SAS data set.

grammar
the rules that define the syntax of the statements used to invoke your procedure.

grammar function
the source code module created by the USERPROC procedure when you use a grammar source module as input. This function is created in the same language that you use to write your procedure. The grammar function must be compiled after the PROC USERPROC step and called by your procedure.

grammar processor
a program run by the USERPROC procedure to combine your grammar source module with a set of standard grammar rules and create a grammar function. See grammar function and grammar source module.

grammar source module
a file of statements that describe the grammar for a procedure.

IFFC
an acronym for informats, formats, functions, and CALL routines. You use similar steps to develop, compile, and link these four types of SAS user-written modules, so they are often mentioned collectively.

IFFC package
a collection of one to ten user-written IFFCs in a single executable file. All routines in a package must be the same type (all formats, all informats, or all functions and CALL routines).

include library
an aggregate storage location that contains members of source code that can be included into other source code modules. Include libraries can be used with grammar source modules or other language modules.

informat
an instruction the SAS System uses to read raw data values to create variable values. There are two types of informats: informats supplied by SAS software and user-written informats created using the FORMAT procedure in base SAS software or using SAS/TOOLKIT software. See also user-written informat.

int
the default integer type on any host. For most hosts, an int is a long integer.

lexical
an element of a grammar that defines restrictions on the type of input for the statement. Lexicals are primitive word types such as numbers or names.

libref
the name temporarily associated with a SAS data library. You assign a libref with a LIBNAME statement or with operating system control language.

link
to create an executable module from one or more object modules. On some systems, the input to the linking process can include other executable modules as well as object modules.

long
a variable type that contains fixed-binary, long-integer data.

lookup data set
the data set the parser uses to validate the list of variables specified in a statement.

macro library
an aggregate storage location that contains terms that expand into other values or expressions.

member-level access
the access to a SAS data library or other file that permits only one user to use a member (such as a SAS data set) at a time. See also record-level access.

memory pool
an area of memory set aside for use by the SAS System or a specific SAS procedure. Memory pools are allocated by calling the SAS_XMPOOLC routine. Portions of the memory pool can be used to store data by issuing other function calls.

native language
the programming language you use to write a SAS procedure. SAS/TOOLKIT software permits you to use C, FORTRAN, PL/I, or IBM 370 assembler language.

NULL pointer
a pointer with a value of 0.

numeric format
instructions to the SAS System to write numeric variable values using a specific pattern.

numeric informat
instructions to the SAS System to read numeric data values using a specific pattern to create numeric variable values.

object module
a file of compiled code. This is an intermediate file between the source module and the executable module. The object module contains the output of a compiler and is input to a linker to create an executable module.

observation number
a number indicating the relative position of an observation in a SAS data set when you read the entire data set sequentially. This number is not stored internally. See also record ID.

operator
a symbol in a grammar that indicates the relationship between the other components of a rule.

parameter
an alias for parameter argument.

parameter argument
a SAS argument in the format *keyword=value*.

parse
the process of breaking a SAS source statement into its component parts so it can be analyzed according to the rules of the grammar.

parser
a part of the SAS System that analyzes statements entered by the user according to grammar rules passed to it.

pointer
a data type available in some languages that contains the address of a data value in memory.

preprocessor
a program that reads a source code file, substitutes values for specially defined symbols, and performs other necessary changes to the source code before passing the altered source code to a compiler.

procedure
a program written in a language other than the SAS language that interfaces with the SAS System and is accessed with a PROC statement. Procedures perform a variety of tasks, including producing reports, managing files, and analyzing data. Many procedures are included with the SAS System; these are usually called SAS procedures. In addition, users can write their own procedures using SAS/TOOLKIT software; these are usually called user-written procedures. See also user-written procedure.

procedure output file
an external file that contains the result of the analysis or the report produced. Most procedures write output to the procedure output file by default. Reports produced by DATA steps that contain a FILE statement with the PRINT destination also go to this file.

production
a single rule in the grammar. A production can be broken into two components, the production name and the rule, as illustrated here:

production-name=rule,

program constants object file
a file created by running the USERPROC procedure with the input file containing the source code for a procedure, format, informat, function, or CALL routine. The program constants object file must be linked to the compiled module for you to use the SAS module with the SAS System.

radix-addressable
a characteristic of a data set that allows you to transform the internal record ID for an observation into an observation number.

record ID
an internal, unique identifier for an observation in a data set. The data set must be radix-addressable to convert the record ID to an observation number.

record-level access
the access to a SAS data set or other file that permits more than one user to access the SAS data set or file at a time. Only one user can use a single observation or record of the file at a time, but other users can access other observations or records in the same file.

return code
a value returned by a program to indicate whether the requested action has been successfully completed.

rich pointer
a pointer type that includes the length in the first byte and the address in the remaining 3 bytes. Rich pointers can be used only under CMS and MVS in releases prior to the XA versions of those operating systems.

rid
an abbreviation for record ID. See record ID.

SAS data file
a SAS data set that is implemented in a form that contains both the data values and the descriptor information. SAS data files have the type DATA.

SAS data library
a collection of SAS files accessed by the same library engine and recognized as a logical unit by the SAS System.

SAS data set
descriptor information and its related data values organized as a table of observations and variables that can be processed by the SAS System. A SAS data set can be either a SAS data file or a SAS data view.

SAS data view
a SAS data set that is implemented in a form that obtains the descriptor information or data values, or both, from other files. Only the information necessary to derive the descriptor information or retrieve the data values is stored in the file of member type VIEW.

SAS date value
an integer representing a date in the SAS System. The integer represents the number of days between January 1, 1960, and another specified date. (For example, the SAS date value 366 represents the calendar date January 1, 1961.)

SAS datetime value
an integer representing the date and time in the SAS System. The integer represents the number of seconds between midnight January 1, 1960, and another specified date and time. (For example, the SAS datetime value for 9:30 a.m., June 5, 1989, is 928661400.)

SAS file
a specially structured file that is created, organized, and, optionally, maintained by the SAS System. A SAS file can be a SAS data set, a catalog, a stored program, or an access descriptor.

SAS invocation
the calling or start-up of the SAS System by an individual user through execution of the SAS command.

SAS keyword
a literal that is a primary part of the SAS language. Keywords are statement names, function names, macro names, and macro function names.

SAS log
a file that contains the SAS statements you have submitted, messages about the execution of your program, and in some cases, output from the DATA step and from certain procedures.

SAS name
a name whose construction follows certain rules and that can appear in a SAS statement (for example, names of variables and SAS data sets). SAS names can be up to eight characters long. The first character must be a letter or an underscore. Subsequent characters can be letters, numbers, or underscores. Blanks and special characters (except the underscore) are not allowed.

SAS program
a series of SAS statements that, taken together, guide the SAS System through a process or series of processes.

SAS Software Consultant
an individual at your computing installation who is designated as a support person for SAS software users at the installation. The consultant can help you with questions about using SAS software.

SAS Software Representative
an individual at your computing installation who is designated as SAS Institute's contact for information on new and existing software. The representative receives any distribution package of software from the Institute. For SAS sites running DOS or OS/2, the representative coordinates distribution of software diskettes within the site. The representative can help you with questions about installing and configuring SAS software.

SAS statement
a string of SAS keywords, SAS names, and special characters and operators ending in a semicolon that instructs the SAS System to perform an operation or gives information to the SAS System.

SAS Supervisor
a part of the SAS System that manages the other parts of the system.

SAS system option
an option that affects the processing of the entire SAS program or interactive SAS session from the time the option is specified until it is removed. Examples of items controlled by SAS system options include the appearance of SAS output, the handling of some of the files used by the SAS System, the use of system variables, the processing of observations in SAS data sets, the features of SAS System initialization, the SAS System's interface with your computer hardware, and the SAS System's interface with the host operating system.

SAS time value
an integer representing a time in the SAS System. The integer represents the number of seconds between midnight of the current day and another specified time value. (For example, the SAS time value for 9:30 a.m. is 34200.)

SAS_X routines
a library of routines included with SAS/TOOLKIT software that provides capabilities for performing I/O on SAS data sets, dynamically allocating memory, printing output to the SAS log or procedure output file, and handling other specialized needs in SAS procedures.

SAS_Z routines
a library of routines included with SAS/TOOLKIT software that provides capabilities that replace native language functions or that extend the native language capabilities.

scatter reading
the process of reading an observation from a SAS data set. The variables in the SAS data set are moved from a contiguous storage area (the input buffer) to separate variables defined in your program.

semantic action
an element of a grammar that signals the parser to perform some action on an associated terminal or lexical. Semantic actions always begin with @.

short
a variable type that contains fixed-binary, short-integer data.

source module
a file of statements written in C or some other programming language. The source module must be compiled and linked before it can be executed on a computer.

statement structure
an internal storage area that the SAS System uses to store information about how a procedure is invoked. When a user invokes a SAS procedure, all of the statements and options used with the procedure are stored in the statement structures so the procedure can determine how to process the user's request.

stub grammar
the standardized grammar provided by SAS Institute that is combined with the grammar for your procedure to create the full set of rules used to parse statements entered by the user.

substitution character
a special character embedded in a character string that indicates where to substitute other values when the string is printed. Different substitution characters are used for each data type.

syntax
a set of rules specifying proper construction of statements or commands.

syntax rules
the requirements defined in a grammar that describe the permitted forms of a SAS statement.

terminal
an element of a grammar that the user specifies to invoke a statement or argument. A terminal is a character string. The terminals you define in your grammar can be any literal string or set of strings, but when the procedure, statement, or option is invoked, the user must specify one of the exact strings you defined. Terminal names are always enclosed in double quotation marks. When the user specifies a terminal, the actions that are associated with the terminal in the grammar are performed.

token
the unit in the SAS language or the macro language into which input must be broken in order to be processed by the SAS System. Tokens (also called words) include items that look like English words (such as variable names) and items that do not (such as mathematical operators and semicolons).

user-written CALL routine
a CALL routine written in C, PL/I, FORTRAN, or IBM 370 assembler using SAS/TOOLKIT software. See also CALL routine.

user-written format
a format you define with the FORMAT procedure or with C, PL/I, FORTRAN, or IBM 370 assembler using SAS/TOOLKIT software. See also format.

user-written function
a function written in C, PL/I, FORTRAN, or IBM 370 assembler using SAS/TOOLKIT software. See also function.

user-written informat
an informat you define with the FORMAT procedure or with C, PL/I, FORTRAN, or IBM 370 assembler using SAS/TOOLKIT software. See also informat.

user-written module
a program written using SAS/TOOLKIT software that can be used with the SAS System to customize SAS processing. User-written modules include SAS procedures, informats, formats, functions, and CALL routines.

user-written procedure
a procedure written in C, PL/I, FORTRAN, or IBM 370 assembler using SAS/TOOLKIT software. See also procedure.

variable list
the list of variables to be processed by a SAS procedure. The variable list can be defined by the user in a VAR statement or other statement, or it can be built by default by the grammar. The variable list is a subset of the data set variable list and contains a list of integers that represent the variable's position in the data set variable list. See also data set variable list.

Index

C

D

M

N

T

V

Y

Z

Special Characters

Your Turn

If you have comments or suggestions about SAS/TOOLKIT software or *SAS/TOOLKIT Software: Usage and Reference, Version 6, First Edition*, please send them to us on a photocopy of this page.

Please return the photocopy to the Publications Division (for comments about this book) or the Technical Support Division (for suggestions about the software) at SAS Institute Inc., SAS Campus Drive, Cary, NC 27513.

SAS/TOOLKIT™ Software

User-written procedures

The SAS® System

User-written informats, formats,

functions, and CALL routines

SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513

ISBN 1-55544-453-9

9 781555 444532