



AS/400™

SC09-1156-00

**Languages:
PL/I
User's Guide and Reference**





AS/400™

SC09-1156-00

Languages:

PL/I

User's Guide and Reference



First Edition (June 1988)

This edition applies to the IBM AS/400™ PL/I (licensed program 5728-PL1), and to any subsequent releases and modifications until otherwise indicated in new editions. Changes are periodically made to the information herein; any such changes will be reported in subsequent revisions.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent program may be used instead.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Publications are not stocked at the address given below. Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to:

IBM Canada Ltd.
Information Development
Department 849
1150 Eglinton Avenue East
North York, Ontario, Canada M3C 1H7

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

AS/400 is a trademark of International Business Machines Corporation.

© Copyright International Business Machines Corporation 1988.

IBM is a registered trademark of International Business Machines Corporation, Armonk, NY.

About This Manual

This *IBM AS/400 Languages: PL/I User's Guide and Reference* manual provides you with the information you need to write, compile, run, test, and maintain PL/I programs on the AS/400¹ System.

Note: This manual describes PL/I as it is used on the AS/400 System. In certain cases, however, it may be useful to also know the System/38 method. These methods are summarized or otherwise described in Appendix F, "Converting from System/38 to the AS/400 System." To find out how to use PL/I in the System/38 Environment, you should refer to *IBM System/38 PL/I Reference Manual and Programmer's Guide, SC09-1051*.

Who Should Use This Manual

This manual is meant for programmers who have some knowledge of and experience in programming with PL/I.

How This Manual is Organized

This manual is divided into two parts. Part 1 is the user's guide to PL/I and Part 2 contains reference information.

- Part 1 is intended to give you the basic information you need to write PL/I programs. Here you will find an introduction to the language and a description of how to actually create, compile, run, test, and debug your PL/I program.
- Part 2 is a catalogue of reference information that you can refer to while using PL/I. Part 2 contains material on program elements, organization and use of data types, data management, and AS/400 files. You will also find material on compiler directives, PL/I statements, references, expressions, and a detailed description of PL/I procedures, subroutines, functions, and pseudovariables.

Following Part 2 is a series of appendixes containing information that might be useful to PL/I users. These include:

- Appendix A, "Compiler Service Information"
- Appendix B, "The AS/400 PL/I Language Summary and Character Set"
- Appendix C, "Valid Combinations of Options for Input/Output Statements"
- Appendix D, "Conditions and Condition Codes"
- Appendix E, "EBCDIC CODES"
- Appendix F, "Converting from System/38 to the AS/400 System"
- Appendix G, "Glossary of Acronyms."

¹ AS/400 is a trademark of International Business Machines Corporation

What You Should Know

This manual also contains a glossary of the AS/400 System and PL/I terms used in this manual that might not be familiar to you, and an alphabetically organized index at the back.

What You Should Know

Before you use this manual, you should be familiar with the information contained in the following IBM AS/400 publications:

- *CL Programmer's Guide*, which contains the basic concepts of the control program functions.
- You should be familiar with your display station (also known as a work station), and its controls. There are also some elements of its display and certain keys on the keyboard that are standard regardless of which software system is currently running at the display station, or which hardware system the display station is hooked up to. Some of these keys are:
 - Cursor movement keys
 - Command keys
 - Field exit keys
 - Insert and delete keys
 - The Error Reset key.

This information is contained in *System Operations: Display Station User's Guide, SC21-9744*.

- You should know how to operate your display station when it is hooked up to the IBM AS/400 System and running the AS/400 System software. This means knowing about the IBM Operating System/400 (OS/400) and the Control Language (CL) to do such things as:
 - Sign on and sign off the display station
 - Interact with displays
 - Use Help
 - Enter control commands and procedure commands
 - Call utilities
 - Respond to messages.

To find out more about this operating system and its control language, refer to:

- *Programming: Control Language Reference, SBOF-0481*
- *Programming: Control Language Programmer's Guide, SC21-8077*
- *Programming: Command Reference Summary, SC21-8076*
- *Programming: System Reference Summary, SC21-8104*
- *Programming: Data Management Guide, SC21-9658*
- You should know how to call and use certain utilities available on the AS/400 System:
 - The Screen Design Aid (SDA) utility used to design and code displays. This information is contained in *Application Development Tools: Screen Design Aid User's Guide and Reference, SC09-1171*.
 - The Source Entry Utility (SEU), which is a full-screen editor you can use to enter and update your source and procedure members. This information is

contained in *Application Development Tools: Source Entry Utility User's Guide and Reference*, SC09-1172.

- You should know how to interpret displayed and printed messages. This information is contained in Chapter 3, "Testing and Debugging PL/I Programs" on page 3-1
- You should be familiar with the PL/I program cycle, how indicators affect the program cycle, and how to code entries on the PL/I specification sheets.

If You Need More Information

You might need to refer to other AS/400 System manuals for specific information about a particular topic. They are listed below:

- *Information Directory*, GC21-9678, which contains a brief description of each manual in the AS/400 library and information on how to order additional publications.
- *Licensed Programs Installation Guide*, SC21-9765, which describes how to install PL/I on your system.
- *System Operations: Operator's Guide*, SC21-8082, which describes how to operate the AS/400 System.
- *Programming: Data Description Specifications Reference*, SC21-9620, which describes data description specifications that are used for describing files.
- *Communications: Distributed Data Management User's Guide*, SC21-9600, which contains information about remote communication for the PL/I programmer.
- *Programming: Data Base Guide*, SC21-9659, which contains a detailed discussion of the AS/400 data base structure. This manual also describes how to use Data Description Specifications (DDS) keywords.
- *Communications: Programmer's Guide*, SC21-9590, which provides information an application programmer needs to write applications that use the AS/400 System communications and the Intersystem Communications Function file.
- *Programming: Graphical Data Display Manager Programming Reference*, SC33-0537, and *Programming: Graphical Data Display Manager Programming Guide*, SC33-0536, which provide guidance on the Graphical Data Display Manager (GDDM) for programmers who need to write graphics applications.
- *System/38 Environment Programmer's Guide and Reference*, SC21-9755, which describes migrating from System/38 and converting to the AS/400 System.
- *Programming: Structured Query Language/400 Reference*, SC21-9608, which provides detailed information on using Structured Query Language (SQL) statements.
- For limitations that may apply to your program but which do not come from PL/I, see the *Programming: Control Language Programmer's Guide*, SC21-8077.

How to Read the Syntax Diagrams

Throughout this manual, syntax is described using the structure outlined below.

- Read the syntax diagrams from left to right, top to bottom, following the path of the line.

The \blacktriangleright — symbol indicates the beginning of a statement.

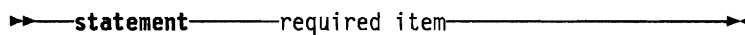
The — \blacktriangleright symbol indicates that the statement syntax is continued on the next line.

The \blacktriangleleft — symbol indicates that a statement is continued from the previous line.

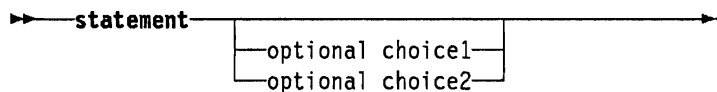
The — \blacktriangleleft symbol indicates the end of a statement.

Diagrams of syntactical units, other than complete statements, start with the \blacktriangleleft — symbol and end with the — \blacktriangleright symbol.

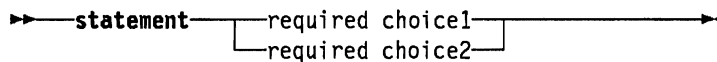
- Required items appear on the main path.



- Optional items appear below the main path. Items will appear in a vertical stack if more than one option is available.

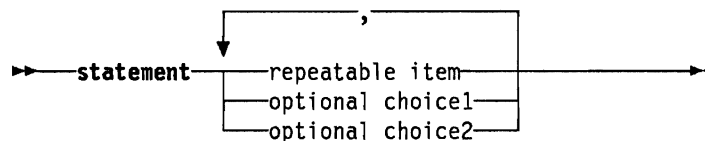


One item of a vertical stack appearing on the main line indicates that at least one option is mandatory.



- An arrow returning to the left above the main line indicates a single item that can be repeated, or an additional option selected from a vertical stack.

The repeat arrow will also indicate any punctuation, such as a comma, that is required between selections.



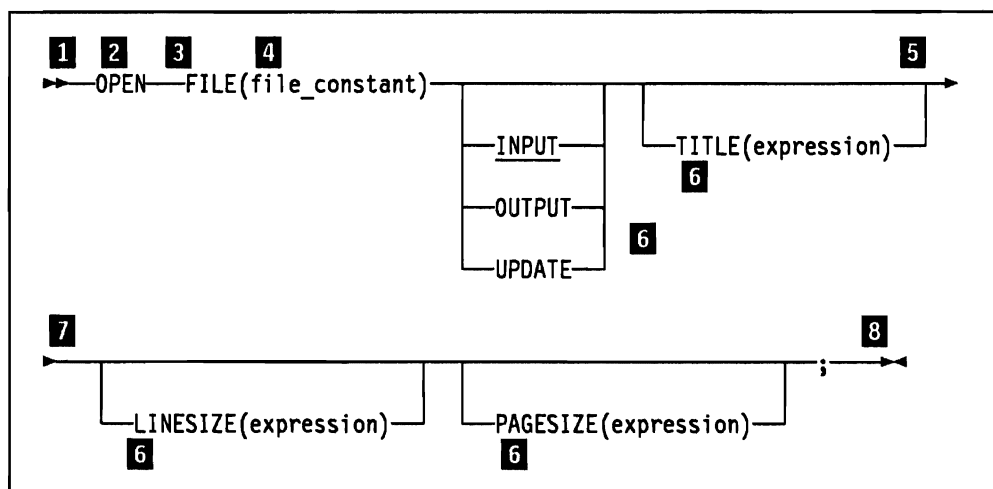
- Enter words in UPPERCASE characters as shown, unless an abbreviation is indicated. Words in lowercase characters represent variable values, and are described following the syntax diagram.
- All round brackets, arithmetic and logical operators, and punctuation must be entered where shown.

- The underscore on any item denotes a default attribute.

Note: For ease of readability, syntax in some areas of the manual will deviate from the above description, as follows:

- Square brackets indicate optional items.
- Vertical bars indicate a choice of items.
- Braces indicate mandatory syntactic expressions.

The following example shows the syntax for the OPEN statement.



The following is the interpretation for the above sample syntax:

- 1 The start of the syntax diagram.
- 2 The keyword OPEN must be entered.
- 3 The keyword FILE(file_constant) must be entered.
- 4 Brackets indicate mandatory part of syntax.
- 5 The syntax is continued at 7 .
- 6 Various options available with the OPEN statement.
- 7 The syntax is continued from 5 .
- 8 The end of the syntax diagram.

Language Extensions

Language extensions are indicated by enclosing the descriptions of the extension in special frames:

Full Language Extension

These brackets indicate language extensions that are part of the complete PL/I (ANSI PL/I X3.53-1976).

End of Full Language Extension

IBM Extension

These brackets indicate language extensions that occur in more than one PL/I compiler, and that are not part of either the complete PL/I (ANSI PL/I X3.53-1976) or the general-purpose subset (ANSI PL/I X3.74-1981).

End of IBM Extension

Industry Standards

The AS/400 PL/I Licensed Program is designed according to the specifications of the following industry standards as understood and interpreted by IBM as of January 1981: American National Standards (ANS) PL/I, X3.53-1976, which is technically identical to International Organization for Standardization (ISO) 6160-1979, and European Computer Manufacturers Association (ECMA) (1976)

The AS/400 PL/I, which is described in this publication, is based on the American National Standards Programming Language PL/I General-Purpose Subset, X3.74-1981, with the following differences:

- Restrictions and omissions from the above subset.
- Extensions based on features of the American National Standard Programming Language PL/I, X3.53-1976.
- Extensions based on common features added by IBM.
- Extensions based on PL/I features added by IBM.

For a complete description of the source of each feature of the language, refer to Appendix B, "The AS/400 PL/I Language Summary and Character Set."

The AS/400 PL/I Licensed Program

The AS/400 PL/I consists of the following:

- A PL/I compiler
- An interface to Source Entry Utility (SEU) for checking PL/I syntax.

What Your AS/400 System Needs to Run PL/I

The AS/400 PL/I Licensed Program (5728-PL1) is run by the IBM Operating System/400 (OS/400) Licensed Program (5728-SS1) on any size AS/400 System that has at least one 1920-character 5250 (or functionally compatible) work station.

The AS/400 PL/I is installed in a separate user library, called QPLI. See the *Licensed Programs Installation Guide, SC21-9765* for information on installing PL/I.



Contents

Part 1. User's Guide

| | |
|--|------|
| Chapter 1. An Introduction to PL/I and the AS/400 System | 1-1 |
| The AS/400 Operating System and Control Language | 1-1 |
| Accessing PL/I on the AS/400 System | 1-1 |
| Chapter 2. Creating, Compiling, and Running Your PL/I Program | 2-1 |
| Creating and Editing the Source Program | 2-1 |
| Using SEU to Create and Edit a Source Program | 2-2 |
| Using SEU to Browse a Compiler Listing | 2-4 |
| Entering SQL Statements into a PL/I Program | 2-5 |
| Compiling Your Source Program Using the CRTPLIPGM Command | 2-5 |
| Completing the First CRTPLIPGM Screen | 2-6 |
| Completing the Second CRTPLIPGM Screen | 2-9 |
| Completing the Third CRTPLIPGM Screen | 2-12 |
| The *DIAGNOSE Option of the GENOPT Parameter | 2-16 |
| Using Compiler Directives | 2-16 |
| Using the %INCLUDE Directive | 2-16 |
| Using the %PAGE Directive | 2-18 |
| Using the %PROCESS Directive | 2-18 |
| Using the %SKIP Directive | 2-19 |
| Compiler Output | 2-19 |
| Running the Program | 2-22 |
| Interrupting or Ending the Running of a Compiled Program | 2-22 |
| Abnormal Program Ending | 2-22 |
| Interlanguage Calls | 2-23 |
| Calling a Non-PL/I Program | 2-23 |
| Calling a PL/I Program from a Non-PL/I Program | 2-24 |
| Chapter 3. Testing and Debugging PL/I Programs | 3-1 |
| Using, Displaying, and Printing Messages | 3-1 |
| Using Messages | 3-1 |
| Displaying and Printing Messages | 3-3 |
| Using a Test Library | 3-3 |
| Using Breakpoints | 3-5 |
| Example of Using Breakpoints | 3-5 |
| Considerations for Using Breakpoints | 3-6 |
| Using a Trace | 3-7 |
| Example of Using a Trace | 3-7 |
| Considerations When Using a Trace | 3-8 |
| Using Debug | 3-10 |
| PL/I Storage | 3-10 |
| Calling Levels | 3-10 |
| Scoping of Names | 3-11 |
| Fully Qualified Names | 3-11 |
| PL/I Pointers | 3-12 |

| | |
|---|------|
| Floating Point Variables | 3-12 |
| Changing Varying Length Strings | 3-12 |
| Specifying Variables by ODV Number | 3-13 |
| Displaying Level Numbers | 3-13 |
| References to Static Variables | 3-13 |
| Determination of Active Blocks in a Program | 3-13 |
| Using PLIDUMP | 3-14 |
| Example of Using PLIDUMP | 3-15 |
| Error Dump Option Screen | 3-16 |
| Using PLIIOFDB and PLIOPNFDB | 3-17 |
| Using ON Conditions | 3-17 |

Part 2. Reference

| | |
|---|------|
| Chapter 4. Program Elements and Organization | 4-1 |
| Characters That are Used in PL/I | 4-1 |
| PL/I Program Structure | 4-1 |
| Statements and Directives | 4-1 |
| Elements of a PL/I Statement | 4-2 |
| Program Organization | 4-6 |
| Programs | 4-6 |
| Blocks | 4-6 |
| Internal and External Procedures | 4-9 |
| Begin-Blocks | 4-11 |
| Names | 4-12 |
| Explicit Declaration of a Name | 4-13 |
| Contextual Declaration of a Name | 4-13 |
| Multiple Declarations of Names | 4-14 |
| Scopes of Names | 4-14 |
| Chapter 5. PL/I Data Organization and Use | 5-1 |
| DATA ORGANIZATION | 5-1 |
| Using Arrays and the Dimension Attribute | 5-1 |
| Using Structures and Level Numbers | 5-3 |
| Arrays of Structures | 5-5 |
| Performance Considerations with Large Aggregates | 5-7 |
| Data Alignment and the Alignment Attributes | 5-7 |
| Data Mapping | 5-9 |
| Scalar Data Mapping | 5-10 |
| Array Mapping | 5-11 |
| Structure Mapping | 5-12 |
| STORAGE CONTROL | 5-15 |
| Using the STATIC Attribute | 5-16 |
| Using the INITIAL Attribute | 5-17 |
| Using the AUTOMATIC Attribute | 5-18 |
| Using the BASED Attribute | 5-19 |
| Based Variable Reference and Pointer Qualification | 5-20 |
| ALLOCATE Statement for Based Variables | 5-22 |
| FREE Statement for Based Variables | 5-23 |
| Data Assignment | 5-24 |
| String Data Assignment | 5-25 |

| | |
|---|------|
| Arithmetic Data Assignment | 5-26 |
| Data Conversion | 5-27 |
| Built-In Conversion Functions | 5-28 |
| Calculating String Length and Precision | 5-29 |
| Conversion Rules | 5-29 |
| Truncation of Floating-Point Data | 5-34 |
| Examples of Data Conversion | 5-34 |
| Chapter 6. AS/400 PL/I File and Record Management | 6-1 |
| File Management | 6-1 |
| File Independence | 6-1 |
| Device Independence | 6-1 |
| System Override Considerations | 6-2 |
| Security | 6-3 |
| Types of Files | 6-4 |
| Data Base Files | 6-4 |
| File Locking | 6-6 |
| Record Locking | 6-7 |
| DEVICE Files | 6-8 |
| DISPLAY Files | 6-8 |
| Other Types of Device Files | 6-9 |
| Using Record Formats | 6-11 |
| Externally Described Record Formats | 6-11 |
| Program-Described Files | 6-12 |
| Chapter 7. File Declaration and Input/Output | 7-1 |
| The ENVIRONMENT Attribute | 7-1 |
| File Organization Options | 7-2 |
| File Locking Options | 7-3 |
| Key Options | 7-3 |
| CTLASA Option | 7-4 |
| BUFSIZE (integer_constant) Option | 7-5 |
| DESCRIBED Option | 7-6 |
| Commitment Control Option | 7-6 |
| Blocking Option | 7-7 |
| NOINDARA Option | 7-8 |
| Opening and Closing Files | 7-11 |
| Scoping of Open Files (File Sharing) | 7-11 |
| Considerations for Opening a Print Stream File | 7-12 |
| Considerations for Opening a Non-Print Stream File | 7-12 |
| Considerations for Opening SYSPRINT | 7-12 |
| Considerations for File Closing after an Error | 7-13 |
| The OPTIONS Option of Record Data Transmission Statements | 7-13 |
| RECORD Parameter | 7-15 |
| KEYSEARCH Parameter | 7-17 |
| POSITION Parameter | 7-17 |
| NBRKEYFLDS Parameter | 7-18 |
| INDICATORS Parameter | 7-19 |
| MODIFIED Parameter | 7-20 |
| Chapter 8. Using AS/400 Files | 8-1 |

| | |
|--|-------------|
| Using Data Base Files | 8-1 |
| Externally Described Data Base Files | 8-2 |
| Program-Described Data Base Files | 8-2 |
| Data Description Specifications | 8-2 |
| Using Display Files | 8-22 |
| Externally Described Display Files | 8-23 |
| Example of Using a Display File | 8-23 |
| Example of Using a Subfile for Displaying Data | 8-29 |
| Example of Using Indicators | 8-43 |
| Using Device Files | 8-49 |
| Externally Described Device Files | 8-50 |
| Program-Described Device Files | 8-50 |
| Example of Using a Printer File | 8-50 |
| Using STREAM Files | 8-55 |
| Example of Using a Stream File | 8-56 |
| Commitment Control | 8-58 |
| Using the COMMITTABLE Option | 8-59 |
| Using the PLICOMMIT Built-In Subroutine | 8-59 |
| Using the PLIROLLBACK Built-In Subroutine | 8-60 |
| Using PLICOMMIT and PLIROLLBACK | 8-61 |
| Examples Using Commitment Control | 8-64 |
| Using the %INCLUDE Directive for External File Descriptions | 8-73 |
| Using the %INCLUDE Directive with Externally Described Files | 8-75 |
| Using the %INCLUDE Directive with Program-Described Files | 8-76 |
| Using the %INCLUDE Directive with Display Files | 8-76 |
| DDS to PL/I Mapping | 8-77 |
| DDS Features You Can Use in Your PL/I Program | 8-78 |
| Sample Program Showing Use of DDS Features | 8-79 |
| | |
| Chapter 9. References and Expressions | 9-1 |
| Operational Expressions | 9-4 |
| Arithmetic Operations | 9-5 |
| Bit Operations | 9-9 |
| Comparison Operations | 9-10 |
| Concatenation Operations | 9-14 |
| Combinations of Operations | 9-14 |
| | |
| Chapter 10. Condition Handling Statements | 10-1 |
| Specifiable Conditions in ON and SIGNAL Statements | 10-1 |
| Unspecifiable Conditions | 10-2 |
| Established Action | 10-2 |
| Implicit Action | 10-2 |
| ON Statement | 10-2 |
| Running an On-Unit | 10-3 |
| Scope of the Established Action | 10-4 |
| Scope of Values of Condition Handling Built-In Functions | 10-4 |
| SIGNAL Statement | 10-4 |
| Example of Use of Conditions | 10-5 |
| | |
| Chapter 11. Input and Output Statements | 11-1 |
| Input and Output | 11-1 |

| | |
|--|-------|
| Files | 11-2 |
| AS/400 Files | 11-2 |
| Use of the File Attributes | 11-3 |
| File Name | 11-3 |
| Type Of Data Transmission | 11-3 |
| Direction of Data Transmission | 11-3 |
| Opening and Closing Files | 11-4 |
| OPEN Statement | 11-4 |
| CLOSE Statement | 11-8 |
| Record Data Transmission | 11-8 |
| Use of File Description Attributes | 11-9 |
| Data Transmission Statements | 11-10 |
| Options of Record Data Transmission Statements | 11-16 |
| Stream Data Transmission | 11-22 |
| File Description Attributes | 11-23 |
| Data Transmission Statements | 11-23 |
| Options of Stream Data Transmission Statements | 11-26 |
| Data Specifications | 11-27 |
| Data Lists | 11-27 |
| Format Lists | 11-28 |
| Format Items | 11-31 |
| Transmission of Array Elements and Structure Fields | 11-41 |
| Print Files | 11-42 |
| SYSIN File | 11-44 |
| SYSPRINT File | 11-44 |
| | |
| Chapter 12. Declaring Names and Attributes of Variables | 12-1 |
| The DECLARE Statement | 12-1 |
| Factoring of Attributes | 12-2 |
| Classification of Attributes | 12-2 |
| Required Attributes | 12-5 |
| Implied Attributes | 12-5 |
| File Attributes | 12-6 |
| Data Types | 12-8 |
| Problem Data Attributes | 12-9 |
| Coded Arithmetic Data Attributes | 12-10 |
| String Data Attributes | 12-15 |
| PICTURE Data Attribute | 12-19 |
| Digit and Decimal Point Characters | 12-22 |
| Zero Suppression Characters | 12-23 |
| Insertion Characters | 12-23 |
| Sign and Currency Characters | 12-24 |
| Credit and Debit Characters | 12-26 |
| Digit and Signed Character | 12-26 |
| Program Control Data Attributes | 12-30 |
| POINTER Attribute | 12-30 |
| LABEL Attribute | 12-31 |
| ENTRY Attribute | 12-32 |
| BUILTIN Attribute | 12-36 |
| VARIABLE Attribute | 12-37 |
| Aggregate Data Declarations | 12-38 |

| | |
|--|-------|
| Arrays and the Dimension Attribute | 12-38 |
| Structures and Level Numbers | 12-39 |
| Alignment Attributes | 12-40 |
| Scope Attributes | 12-40 |
| Storage Attributes | 12-41 |
| AUTOMATIC Attribute | 12-41 |
| BASED Attribute | 12-42 |
| STATIC Attribute | 12-42 |
| INITIAL Attribute | 12-42 |
| | |
| Chapter 13. General PL/I Statements | 13-1 |
| Assignment Statement | 13-1 |
| Examples of Assignment Statements | 13-2 |
| BY NAME ASSIGNMENT | 13-3 |
| DO Statement | 13-5 |
| Examples of DO Statements | 13-8 |
| END Statement | 13-10 |
| GO TO Statement | 13-11 |
| IF Statement | 13-13 |
| Examples of IF Statements | 13-14 |
| ITERATE Statement | 13-14 |
| Example of the ITERATE Statement | 13-15 |
| LEAVE Statement | 13-15 |
| Examples of LEAVE Statements | 13-16 |
| Null Statement | 13-17 |
| Examples of Null Statements | 13-17 |
| SELECT, WHEN, and OTHERWISE Statements | 13-18 |
| Examples of Select-Groups | 13-19 |
| STOP Statement | 13-20 |
| | |
| Chapter 14. Procedures, Subroutines, and Functions | 14-1 |
| Defining a Procedure | 14-1 |
| PROCEDURE Statement | 14-2 |
| RETURN Statement | 14-4 |
| Calling a Procedure | 14-4 |
| Function Reference | 14-4 |
| CALL Statement | 14-7 |
| Association of Arguments and Parameters | 14-9 |
| Recursive Procedures | 14-11 |
| | |
| Chapter 15. Built-In Functions, Subroutines, and Pseudovariabes | 15-1 |
| Declaring a Built-In Function or Built-In Subroutine | 15-1 |
| Built-In Functions | 15-1 |
| Classification of Built-In Functions | 15-2 |
| Built-In Subroutines | 15-4 |
| Pseudovariabes | 15-5 |
| Aggregate Arguments | 15-5 |
| Empty Argument Lists | 15-5 |
| Descriptions of Built-In Functions, Subroutines and Pseudovariabes | 15-5 |
| ABS(x) | 15-5 |
| ACOS(x) | 15-6 |

| | |
|---------------------------------|-------|
| ADDR(x) | 15-6 |
| ASIN(x) | 15-6 |
| ATAN(x,y) | 15-7 |
| ATAND(x,y) | 15-7 |
| ATANH(x) | 15-7 |
| BINARY(x,p[,0]) | 15-8 |
| BIT(x,y) | 15-8 |
| CHARACTER(x,y) | 15-8 |
| COPY(x,y) | 15-8 |
| COS(x) | 15-9 |
| COSD(x) | 15-9 |
| COSH(x) | 15-9 |
| DATE[()] | 15-9 |
| DECIMAL(x,p[,q]) | 15-9 |
| DIMENSION(x,y) | 15-10 |
| DIVIDE(x,y,p[,q]) | 15-10 |
| EXP(x) | 15-11 |
| FIXED(x,p[,q]) | 15-11 |
| FLOAT(x,p) | 15-11 |
| HBOUND(x,y) | 15-11 |
| INDEX(x,y) | 15-12 |
| LBOUND(x,y) | 15-12 |
| LENGTH(x) | 15-12 |
| LINENO(x) | 15-12 |
| LOG(x) | 15-12 |
| LOG2(x) | 15-13 |
| LOG10(x) | 15-13 |
| MAX(x1,x2) | 15-13 |
| MIN(x1,x2) | 15-13 |
| MOD(x,y) | 15-14 |
| NULL[()] | 15-14 |
| ONCODE[()] | 15-14 |
| ONFILE[()] | 15-15 |
| ONKEY[()] | 15-15 |
| PLICOMMIT Built-In Subroutine | 15-15 |
| PLIDUMP Built-In Subroutine | 15-16 |
| PLIIOFDB Built-In Subroutine | 15-16 |
| PLIOPNFDB Built-In Subroutine | 15-17 |
| PLIRCVMSG Built-In Subroutine | 15-18 |
| PLIRETC Built-In Subroutine | 15-18 |
| PLIRETV[()] | 15-19 |
| PLIROLLBACK Built-In Subroutine | 15-20 |
| PLISHUTDN[()] | 15-20 |
| ROUND(x,y) | 15-20 |
| SAMEKEY(x) | 15-21 |
| SIGN(x) | 15-21 |
| SIN(x) | 15-22 |
| SIND(x) | 15-22 |
| SINH(x) | 15-22 |
| SQRT(x) | 15-22 |
| STORAGE(x) | 15-22 |

| | |
|--|------------|
| SUBSTR(x,y[,z]) | 15-23 |
| SUBSTR(x,y[,z]) Pseudovvariable | 15-23 |
| TAN(x) | 15-24 |
| TAND(x) | 15-24 |
| TANH(x) | 15-24 |
| TIME[()] | 15-24 |
| TRANSLATE(x,y[,z]) | 15-24 |
| TRUNC(x) | 15-25 |
| UNSPEC(x) | 15-25 |
| UNSPEC(x) Pseudovvariable | 15-25 |
| VERIFY(x,y) | 15-26 |
| Appendix A. Compiler Service Information | A-1 |
| Compiler Overview | A-1 |
| Compiler Organization | A-3 |
| Compiler Phases | A-3 |
| Intermediate Text | A-5 |
| Compiler Segments | A-5 |
| Formatters and Intermediate Text | A-9 |
| Error Message Organization | A-9 |
| Compiler Debugging Options | A-10 |
| Examples of Using Compiler Debugging Options | A-10 |
| Using the SERVICE Parameter | A-15 |
| Quantitative Limits of Compiler | A-19 |
| Appendix B. The AS/400 PL/I Language Summary and Character Set | B-1 |
| The PL/I Character Set | B-14 |
| Extralingual Characters | B-15 |
| Appendix C. Valid Combinations of Options for Input/Output Statements | C-1 |
| Data Base Files with CONSECUTIVE organization | C-2 |
| Data Base Files with INDEXED organization | C-3 |
| Display Files with INTERACTIVE organization | C-4 |
| Subfiles with INTERACTIVE organization | C-5 |
| Display Files with CONSECUTIVE organization | C-6 |
| Inline Files with CONSECUTIVE organization | C-7 |
| Printer Files with CONSECUTIVE organization | C-7 |
| Tape and Diskette Files with CONSECUTIVE organization | C-8 |
| Communications and BSC Files with INTERACTIVE and | C-9 |
| Appendix D. Conditions and Condition Codes | D-1 |
| Conditions | D-1 |
| Condition Codes | D-6 |
| Appendix E. EBCDIC CODES | E-1 |
| Appendix F. Converting from System/38 to the AS/400 System | F-1 |
| Your Choice of Two Environments: AS/400 System or the System/38 Environment | F-1 |
| Compiling in the System/38 Environment | F-1 |
| Writing Programs in the System/38 Environment | F-2 |

| | |
|---|----------------|
| Writing Programs in the System/38 Environment | F-2 |
| Using the %INCLUDE Directive | F-2 |
| Using the %INCLUDE Directive for External File Descriptions | F-2 |
| Syntax of TITLE Parameter of the OPEN Statement | F-3 |
| Appendix G. Glossary of Abbreviations | G-1 |
| Glossary of Terms | GLOSS-1 |
| Index | X-1 |



Part 1. User's Guide

Part 1 is a user's guide. It contains the basic information you need to program in AS/400 PL/I. This information is organized sequentially to allow you to read through it and develop an understanding of PL/I programming: how to create, compile, run, test, and debug your program.

The user's guide is organized into:

- Chapter 1, "An Introduction to PL/I and the AS/400 System"
- Chapter 2, "Creating, Compiling, and Running Your PL/I Program"
- Chapter 3, "Testing and Debugging PL/I Programs."



Chapter 1. An Introduction to PL/I and the AS/400 System

This chapter is a brief introduction to using AS/400 PL/I. The topics include:

- The AS/400 Operating System and Control Language
- Accessing PL/I from OS/400

The AS/400 Operating System and Control Language

The AS/400 Operating System

The operating system that controls all your interactions with the AS/400 System is called Operating System/400 (OS/400). From your display work station, OS/400 allows you to:

- Sign on and sign off the AS/400 System
- Interact with the displays
- Use Help
- Enter control commands and procedures
- Respond to messages
- Manage files
- Call up other utilities and run other programs.

The AS/400 Control Language

You interact with the AS/400 System by entering or selecting Control Language (CL) commands.

The AS/400 CL commands you will be using most often with PL/I are:

- STRSEU to call up the Source Entry Utility (SEU), a full-screen editor that can be used to enter PL/I program code
- CRTPLIPGM to compile PL/I source programs
- CALL program-name to run a PL/I program
- CALL QCL to access the System/38 Environment
- RETURN to exit from System/38 Environment.

The Control Language and all its commands are described in detail in the *Programming: Control Language Reference*.

Accessing PL/I on the AS/400 System

When you start working on the AS/400 System, you will see the following screen.

ACCESSING PL/I on the AS/400 System

```

                                     Sign On
                                     System . . . . . : XXXXXXXX
                                     Subsystem . . . . . : XXXXXXXXXXXX
                                     Display . . . . . : XXXXXXXXXXXX

User ID . . . . . : _____
Password . . . . . : _____
Program/procedure . . . . . : _____
Menu . . . . . : _____
Current library . . . . . : _____

_____ error message line-----
                                     © COPYRIGHT IBM CORP. 1988
```

Figure 1-1. The AS/400 System Sign-on Screen

The following screen appears, when you enter your ID and password, and you can begin working on the AS/400 System.

```

MAIN                               AS/400 Main Menu           System: xxxxxxxx

Select one of the following:

    1. User tasks
    2. Office tasks
    3. General system tasks
    4. Files, libraries, and folders
    5. Programming
    6. Communications
    7. Define or change the system
    8. Problem handling
    9. Display a menu

   90. Sign off

Selection or command
===> _____

F3=Exit   F4=Prompt   F9=Retrieve   F12=Previous   F13=User Support
F23=Set initial menu

                                     © COPYRIGHT IBM CORP. 1988
```

Figure 1-2. AS/400 Main Menu Screen

To begin working in PL/I, enter or select the appropriate CL command.

Chapter 2. Creating, Compiling, and Running Your PL/I Program

To run a program, you must enter and store it on the AS/400 System as a source file, and then compile it. You can connect programs written in different languages, including PL/I, and run the PL/I program as part of a system of programs. The chapter describes:

- Creating and editing the source program
- Compiling your source program using the CRTPLIPGM command
- Using compiler directives
- Running the program
- Compiler output
- Interlanguage calls.

Creating and Editing the Source Program

You can enter your source program onto the system interactively, by using the Source Entry Utility (SEU). Enter the CL command STRSEU (Start SEU) to call SEU.

For a description of how to use the STRSEU command, refer to the *SEU User's Guide and Reference*.

You can enter your source program onto the system in batch mode (for example, from diskettes) by using the OS/400 copy or spooling functions. For more information on how to use the copy and spooling functions for batch entry, refer to the *Programming: Data Management Guide*.

The first procedure in creating your program is to name the file that will store the source program. The AS/400 file naming convention that you use to do this is **library-name/file-name**.

Note: A PL/I program that is entered in the System/38 Environment should also be compiled and run in the System/38 Environment. The program can access any file unless the file name contains characters other than A-Z, 0-9, #, @, and _. Because this restriction does not apply to AS/400 file names, you may not be able to use a AS/400 file from the System/38 Environment.

On the AS/400 System, you can use upper or lowercase characters in a file name or a member name. However, all lowercase characters are converted to uppercase in the System/38 Environment.

CREATING AND EDITING PROGRAMS

Using SEU to Create and Edit a Source Program

SEU provides you with a screen that you can use to enter your source program, and a PL/I syntax checker that checks each line for errors as you enter it. There are also three screens that you can use for various functions on a file you are editing. The screens are:

- The Edit Services Screen shown in Figure 2-1 on page 2-3
- The Find/Change Services Screen shown in Figure 2-2 on page 2-3
- The Browse/Copy Services Screen shown in Figure 2-3 on page 2-4.

The PL/I Syntax Checker

The following commands allow you to use the PL/I syntax checker.

- The CL command `STRSEU TYPE(PLI)` accesses SEU with the PL/I syntax checker in effect. You can also select the `TYPE(PLI)` parameter from the SEU edit services screen (see Figure 2-1 on page 2-3).
- The CL command `STRSEU TYPE(TEXT)` accesses SEU as an editor only; no syntax checker is in effect.
- The CL command `STRSEU TYPE(PLI38)` calls up the System/38 PL/I syntax checker. The syntax rules for System/38 PL/I apply.

If you use the PL/I syntax checker while entering your source program, pressing Enter at any time automatically processes the syntax checker on any line that has been changed and on any new lines that have been added to the screen. Any statement that contains a syntax error is then shown in reverse image, and an error message appears on the screen telling you what is wrong with the statement. When you correct the error and press Enter, the error message is taken off the screen and the normal image of the statement is restored.

The syntax checker only checks individual statements, independently of preceding statements. Therefore no errors based on relationships with other statements are detected. For instance, if you declare `WAGETOT` at the beginning of your program and misspell the variable name in a later statement

```
WAGETOTAL = CURMONTHTOTAL + YDOTAL;
```

no error is detected. Similarly, if you make an error in nesting loops and code too many `END` statements, the syntax checker cannot detect the error. This type of error is found when you compile the source program.

| Edit Services | | |
|---|------|----------------------------|
| Type choices, press Enter. | | |
| Amount to roll | 1 | 1=Half page 2=Full page |
| Uppercase input only | Y | Y=Yes, N=No |
| Tabs on | N | Y=Yes, N=No |
| Increment of insert record | 0.01 | 0.01 to 999.99 |
| Source type | PLI | |
| Syntax checking: | | |
| When added/modified | Y | Y=Yes, N=No |
| From sequence number | | 0000.00 to 9999.99 |
| To sequence number | | 0000.00 to 9999.99 |
| Left margin | 2 | 1 to 80 |
| Right margin | 72 | 1 to 80 |
| Set records to date | / / | YY/MM/DD or YYMMDD |
| Screen size | 1 | 1=27x132, 2=24x80 |
| F3=Exit F5=Refresh F12=Previous | | |
| F14=Find/Change Services F15=Browse/Copy Services | | |

Figure 2-1. SEU Edit Services Screen

| Find/Change Services | | |
|--|----------|---|
| Type choices, press Enter. | | |
| Find | | |
| Change | | |
| From column number | 1 | 1 to 80 |
| To column number | 80 | 1 to 80 |
| Allow data shift | Y | Y=Yes, N=No |
| Occurrences to process | 1 | 1=Next, 2=All 3=Previous |
| Records to search | 1 | 1=All, 2=Excluded 3=Non-excluded |
| Kind of match | 2 | 1=Same case 2=Ignore case |
| Search for date | 88/11/19 | YY/MM/DD or YYMMDD |
| Compare | | 1=Less than 2=Equal to 3=Greater than |
| F3=Exit F5=Refresh F12=Previous F13=Edit Services | | |
| F15=Browse/Copy Services F16=Find F17=Change | | |

Figure 2-2. SEU Find/Change Services Screen

CREATING AND EDITING PROGRAMS

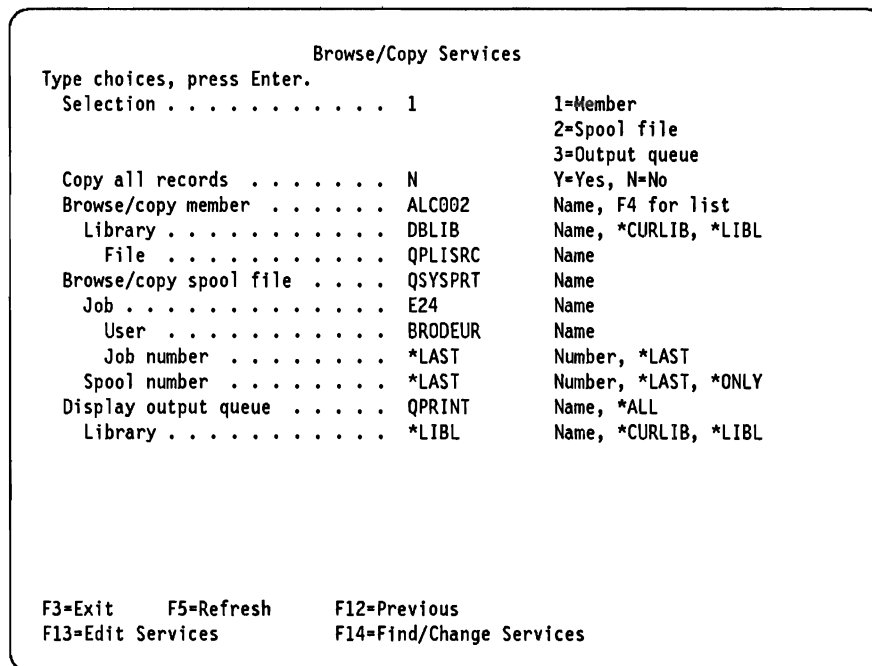


Figure 2-3. SEU Browse/Copy Services Screen

The PL/I syntax checker sets the margins for your source entry to column 2 and column 72. You can see the margin settings by looking at the lower right-hand corner of your file's services screen. For normal PL/I programming this is the standard and desirable setting, but at times you may need to change the setting. For instance, the %PROCESS directive must begin in column 1. %PROCESS is not valid if it begins in any other column. In this case, you should alter the left margin to column 1 so that you can enter the %PROCESS directive correctly. You use the Edit Services screen and change the column number for the left margin from 002 to 001. If you change the margin to 001, you must make sure that the standard PL/I statements in your program do not begin in the first column of the screen.

SEU automatically runs the PL/I syntax checker whenever there are lines that are changed or added on the screen. The new source program is then passed to the syntax checker one statement at a time. Because PL/I source data can be entered in free format and its statements can span more than one line, SEU uses the semicolon to determine the statement boundaries.

Note: The scanning of the semicolon in the backward direction to find the start of a statement may occasionally produce unwanted results when the semicolon is imbedded inside a comment or a string literal, or when the %INCLUDE directive is used in the middle of a statement.

Using SEU to Browse a Compiler Listing

You may use the SEU split-edit display to browse through a compiler listing that is on an output queue. For more information on browsing through a compiler listing, see the *SEU User's Guide and Reference*.

Entering SQL Statements into a PL/I Program

The Structured Query Language (SQL) is a high level data base storage and retrieval language that uses structured techniques. You can place SQL statements into a PL/I program at any point using the SEU to enter the statements.

To enter SQL statement(s) into a PL/I program, you would enter:

```
EXEC SQL sql-statement;
```

Each SQL statement should end with a semicolon and must be on a separate line. The *Programming: Structured Query Language/400 Reference* describes SQL in detail.

Note: SQL statements are recognized and syntax checked by the editor only; not the compiler. They are not processed.

If your program contains SQL statements, you must call the SQL preprocessor before using CRTPLIPGM to compile the source program. Refer to *Programming: Structured Query Language/400 Reference* for a description of how this is done. This is not necessary if the source program has no SQL statements.

Compiling Your Source Program Using the CRTPLIPGM Command

To compile a PL/I source program use the CL command CRTPLIPGM (Create PL/I Program). The compiler checks the syntax of each line of the PL/I source program, and checks relationships among the lines. By selecting options with the CRTPLIPGM command, you can request a program object, a compiler listing, or any of the other options provided. You may use the command directly from OS/400, in a CL program, or in interactive or batch mode.

You can use compiler directives in your PL/I source program to direct some of the operations of the PL/I compiler. Compiler directives allow you to:

- Copy external text into the program
- Copy data description specifications for externally described files into the program
- Control batch compilation
- Control the format of the program listing.

For more information, see "Using Compiler Directives" on page 2-16.

When compiling takes place, an attribute character string is produced that specifies the environment that the program was compiled in. The character string is PLI for the AS/400 System and PLI38 for the System/38 Environment. Other differences in compiling in the System/38 Environment are described in "Compiling in the System/38 Environment" on page F-1.

If the compilation is successful, a message identified by code PLC0005 is sent and the return code is set to zero. If the compilation is not successful, a message identified by code PLC9001 is sent and the return code is set to 2. The CL command

COMPILING SOURCE PROGRAMS

MONMSG (Monitor Message) can be used in a CL program to monitor for these messages.

All object names specified on the CRTPLIPGM command must be composed of alphanumeric characters, the first of which must be alphabetic. The length of the names cannot exceed ten characters. See *Programming: Control Language Reference* for a detailed description of OS/400 object naming rules and for a complete description of CL command syntax.

Completing the First CRTPLIPGM Screen

```
CRTPLIPGM                Create PL/I Program
Type choices, press Enter.
Program . . . . . *PROC      Name, *PROC
Library . . . . . *CURLIB    Name, *CURLIB
Source file . . . . . QPLISRC Name
Library . . . . . *LIBL      Name, *LIBL, *CURLIB
Source member . . . . . *PGH   Name, *PGH
Generation severity level . . . 15      0-29
Text 'description' . . . . . *SRCHBRTXT

F3=Exit  F4=List  F5=Refresh  F10=Additional parameters  F11=Keywords
F12=Previous  F13=Prompter help
```

Figure 2-4. The First CRTPLIPGM Screen

Each parameter on the screen displays a default value. Move the cursor past items where you want the default value to apply. Type over any items where you want to set a different value or option. If you are not sure about the setting of a particular parameter, type a question mark (?) as the first character in that field and press Enter to receive more detailed information. The ? must be followed by a blank.

You must enter values for the library and program name by which the compiled program is known, and the name of the source file that contains the program source.

All other parameters have default values, which you can change if necessary. Press F10 to display additional parameters (see Figure 2-5 on page 2-9). Press F3 to exit without processing the command.

The descriptions of the parameters and options follows (the defaults are underlined and are explained first).

PGM

Specifies the library and program name by which the compiled PL/I program is known. If no library is specified, the created program is stored in the current library. The program must not already exist in the library.

*PROC

The program name is the name of the first external procedure in the compilation. If there is more than one program in the compilation, each subsequent program name is the name of the first external procedure following each %PROCESS directive.

COMPILING SOURCE PROGRAMS

program-name

The name by which the program will be known. This name must match the name of the first external procedure in the compilation.

Note: If the name you enter does not match the name of the first external procedure in the compilation, there will be an unrecoverable error, and the program will not be compiled.

If there is more than one program in the compilation, each subsequent program name is the name of the first external procedure following each %PROCESS directive.

*CURLIB

The current library will be used. If you have not specified a current library, QGPL will be used.

library-name

Enter the name of the library where the compiled program will be stored.

SRCFILE

Specifies the name of the source file that contains the PL/I source program to be compiled.

QPLISRC

The default source file, QPLISRC, contains the PL/I source file to be compiled.

source-file-name

Enter the name of the source file that contains the PL/I source program to be compiled.

Note: The recommended record length of the PL/I source file is 92. If the record length is greater than 92, only the first 92 bytes of each record is used. The record length must not be less than the value of the right margin plus 12.

*LIBL

The system searches the library list to find the library where the source file is located.

*CURLIB

The current library will be used. If you have not specified a current library, QGPL will be used.

library-name

Enter the name of the library where the source file is stored.

SRCMBR

Specifies the name of the member of the source file that contains the PL/I source program to be compiled. This parameter can only be specified if the source file in the SRCFILE parameter is a data base file.

COMPILING SOURCE PROGRAMS

*PGM

Use the name specified by the PGM parameter as the source file member name. The source program will have the same name as the object program. If no program name is specified by the PGM parameter the system uses the first member created in or added to the source file as the source file member name.

source-file-member-name

Enter the name of the member that contains the PL/I source program.

GENLVL

Specifies whether an object program is generated depending on the severity of the errors encountered. A severity level value corresponding to the severity level of the messages produced during compilation can be specified with this parameter. If errors occur in a program with a severity equal to or greater than the value specified in this parameter, the compilation is ended. For example, if you do not want a program generated if you have messages with a severity level of 20 or greater, specify 20 in this parameter.

15

If a severity level value greater than 15 is specified, the program may contain errors that will cause unpredictable results when the compiled program is run.

severity-level

Enter a two-digit number, 01 through 29.

Note: The severity level value of PL/I messages does not exceed 29.

TEXT

Lets the user enter text that briefly describes the program and its function. The text appears whenever the program runs.

*SRCMBRTXT

The text of the source member is used.

*BLANK

No text appears.

'description'

Enter the text that briefly describes the program and its function. The text can be a maximum of 50 characters in length and must be enclosed in apostrophes. The apostrophes are not part of the 50-character string.

If these parameter values are sufficient, press F16 to process the command. Otherwise, press F10 to display additional parameters.

Completing the Second CRTPLIPGM Screen

```

CRTPLIPGM                Create PL/I Program
Type choices, press Enter.
Program . . . . . *PROC      Name, *PROC
Library . . . . . *CURLIB    Name, *CURLIB
Source file . . . . . QPLISRC Name
Library . . . . . *LIBL      Name, *LIBL, *CURLIB
Source member . . . . . *PGM   Name, *PGM
Generation severity level . . . 15      0-29
Text 'description' . . . . . *SRCMBRTXT
                                Additional Parameters
Compiler options . . . . .
                                + for more values
Generation options . . . . .
                                + for more values
                                More...
F3=Exit  F4=List  F5=Refresh  F11=Keywords  F12=Previous  F13=Prompter help
    
```

Figure 2-5. The Second CRTPLIPGM Screen

The additional parameters and their descriptions are listed below. Default values are underlined>.

OPTION

Specifies the options to use when the PL/I source program is compiled. Any or all of the options can be specified in any order. Separate the options with a blank space.

*SOURCE

Produce a source listing, consisting of PL/I program input and all compile-time errors.

*NOSOURCE

A source listing is not produced. If *NOSOURCE is specified the system defaults to *NOXREF.

The acceptable abbreviation for *SOURCE is *SRC, and for *NOSOURCE is *NOSRC.

*XREF

Produce a cross-reference listing between the items declared in your program and the numbers of the statements in your program that refer to these items. If you specify both *ATR and *XREF, the attribute table and cross-reference listing are combined.

*NOXREF

Do not produce a cross-reference listing.

*SREF

Produce a cross-reference listing of only referenced names. Unreferenced names are omitted.

COMPILING SOURCE PROGRAMS

*GEN

Create a program object that can be run after the program is compiled.

*NOGEN

Do not create a program object.

*NOOPTIONS

Do not list options in effect for this compilation.

*OPTIONS

List options in effect for this compilation.

The acceptable abbreviations for *NOOPTIONS is *NOOPT, and for *OPTIONS is *OPT.

*NOAGGREGATE

Do not generate an aggregate table.

*AGGREGATE

Generate an aggregate table. The aggregate table gives the lengths of all arrays and major structures in the source program.

The acceptable abbreviations for *NOAGGREGATE is *NOAGR, and for *AGGREGATE is *AGR.

*NOATTRIBUTES

Do not generate a table of the attributes of the identifiers in the source.

*ATTRIBUTES

Generate a table of the attributes. If you specify both *ATTRIBUTES and *XREF, the attribute table and cross-reference listing are combined. If you specify both *ATTRIBUTES and *SREF, the attribute table and cross-reference listing for referenced names are combined.

The acceptable abbreviations for *ATTRIBUTES is *ATR, and for *NOATTRIBUTES is *NOATR.

*NOSECLVL

Do not list second-level message text for this compilation.

*SECLVL

List second-level message text for this compilation.

GENOPT

Specifies the options used to create the program object: the printing of the IRP (intermediate representation of a program), a cross-reference listing of objects defined in the IRP, and the program template. GENOPT reserves a program patch area, and specifies optimization of a program for more efficient running. These results may be useful if a problem occurs when trying to run the compiled program. Any or all of the options can be specified in any order. Separate the values with a delimiter.

*NOLIST

Do not list the intermediate representation of the program (IRP), associated hexadecimal code, and error messages. If you specify *XREF, *DUMP, or *ATR, a listing will be generated, even if you specify *NOLIST.

*LIST

List the intermediate representation of the program.

*NOXREF

Do not produce a cross-reference listing of all objects defined in the IRP.

*XREF

Produce a cross-reference listing of all objects defined in the IRP. If you specify *XREF, a listing will be generated, even if you specify *NOLIST.

*NOPATCH

Do not reserve space in the compiled program for a program patch area. The program patch area can be used for debugging.

*PATCH

Reserve space in the compiled program for a program patch area.

*NODUMP

Do not list the program template.

*DUMP

List the program template. If you specify *DUMP, a listing will be generated, even if you specify *NOLIST.

*NOATTRIBUTES

Do not list the attributes for the IRP source.

*ATTRIBUTES

List the attributes for the IRP source. If you specify *ATTRIBUTES, a listing will be generated, even if you specify *NOLIST.

The acceptable abbreviations for *NOATTRIBUTES is *NOATR, and for *ATTRIBUTES is *ATR.

*NODIAGNOSE

Do not process program-checking functions at run time. For more information on the functions provided by the *DIAGNOSE option, see "The *DIAGNOSE Option of the GENOPT Parameter" on page 2-16.

COMPILING SOURCE PROGRAMS

*DIAGNOSE

Process program-checking functions at run time.

*NOOPTIMIZE

Do not process program optimization.

*OPTIMIZE

Process program optimization. With *OPTIMIZE the compiler generates a program for more efficient processing and that will possibly require less storage. However, specifying *OPTIMIZE can substantially increase the time required to create the program. Existing object programs may be optimized using the CL command CHGPGM.

If these parameter values are sufficient, press F16 to process the command. Otherwise, roll the screen to display additional parameters.

Completing the Third CRTPLIPGM Screen

| CRTPLIPGM | | Create PL/I Program | |
|------------------------------------|----------|------------------------------|---|
| Type choices, press Enter. | | | |
| Source margins | | | |
| | *SRCFILE | 1-80 | |
| | | 1-80 | |
| Include file | *SRCFILE | Name | |
| Library | | Name, *LIBL, *CURLIB | |
| Print file | QSYSVRT | Name | |
| Library | *LIBL | Name, *LIBL, *CURLIB | |
| Flagging severity | 0 | 0-49 | |
| Replace existing program | *YES | *YES, *NO | |
| User profile | *USER | *USER, *OWNER | |
| Authority | *CHANGE | Name, *CHANGE, *ALL, *USE... | |
| Compiler problem determination | *NO | *NO, *YES | |
| | | | Bottom |
| F3=Exit | F4=List | F5=Refresh | F11=Keywords F12=Previous F13=Prompter help |

Figure 2-6. The Third CRTPLIPGM Screen

The additional parameters and their descriptions are listed below. Default values are underlined>.

MARGINS

Specifies the part of the compiler input record that contains source text.

*SRCFILE

Use the margin values of the file member you specify in the SRCMBR parameter. If the file is of type PLI, the margin values are the values specified on the SEU services display. If the file is of a different type, the margin values are the default values of 2 and 72.

left, right

Enter the values for the left and right margins. The margins must not be less than 1 or more than 80, and the left margin must be smaller than the right margin.

Note: MARGINS does not apply to the %PROCESS directive, which must have a percent sign (%) in column 1.

INCFILE

Specifies the qualified name of the source file that contains member(s) included in the program with the %INCLUDE directive(s).

*SRCFILE

The qualified source file you specify in the SRCFILE parameter contains the source file member(s) specified on any %INCLUDE directive(s) in the program that either specify SYSLIB or do not specify a file name.

source-file-name

Enter the name of the source file that contains the source file member(s) specified on any %INCLUDE directive(s) in the program that either specify SYSLIB or do not specify a file name. The record length of the source file you specify here must be no less than the record length of the source file you specify for the SRCFILE parameter.

*LIBL

The system searches the library list to find the library.

*CURLIB

The current library will be used. If you have not specified a current library, QGPL will be used.

library-name

Enter the name of the library where the source file is located.

PRTFILE

Specifies the name of the file where the compiler listing is placed and the library where the file is located. If you specify a file whose record length is less than 132, information will be lost.

QSYSPRT

If a file name is not specified, the compiler listing is placed in the IBM-supplied file, QSYSPRT. If the file is spooled, the file goes to the QPRINT queue. The file QSYSPRT has a record length of 132.

print-file-name

Enter the name of the file where the compiler listing is directed.

*LIBL

The system searches the library list to find the library.

*CURLIB

The name of the current library. If you have not specified a current library, QGPL will be used.

library-name

Enter the name of the library where the file is located.

COMPILING SOURCE PROGRAMS

FLAG

Specifies the minimum severity level of messages to be listed.

0

All messages are listed.

severity-level

Enter a number that specifies the minimum severity level of the messages that are listed. Messages that have severity levels of the specified level or higher are listed.

REPLACE

Specifies if a new program object will be created when there is an existing program object of the same name in the same library.

*YES

A new program object will be created and any existing program object of the same name in the specified library will be moved to library QRPLOBJ.

*NO

A new program object will not be created if a program object of the same name already exists in the specified library.

USRPRF

Specifies the user profile the compiled PL/I program runs under. This profile controls which objects can be used by the program (including what authority the program has for each object).

*USER

The program runs under the user profile of the program's user.

*OWNER

The program runs under the user profiles of both the program's owner and user. The collective sets of object authority in both user profiles are used to find and access objects while the program is running. Any objects that are created while the program is running are owned by the program's user.

Note: The USRPRF parameter reflects the security requirements of your installation. The security facilities available on the AS/400 System are described in detail in *Programming: Control Language Programmer's Guide* and the *Programming: Control Language Reference*.

AUT

Specifies what authority for the program and its description is being granted to the public. The authority can be altered for all or for specified users after program creation with the CL commands GRTOBJAUT (Grant Object Authority) and RVKOBJAUT (Revoke Object Authority). For further information on these commands and for an expanded description of the AUT parameter, see the *Programming: Control Language Reference*.

*CHANGE

The public has operational rights only for the compiled program. Any user can run the program and debug it but cannot change it.

***USE**

The public can run the program, but cannot debug or change it.

***ALL**

The public has complete authority for the program.

***EXCLUDE**

The public cannot use the program.

authorization-list

Name of an authorization list to which the program is added. For a description of the authorization list and how to create it see the *Programming: Control Language Reference*.

Note: Use the AUT parameter to reflect the security requirements of your installation. The security facilities available on the AS/400 System are described in detail in *Programming: Control Language Programmer's Guide* and the *Programming: Control Language Reference*.

SERVICE

Specifies the use of the compiler problem determination facilities. For a description of how to use these facilities, refer to Appendix A, "Compiler Service Information."

***NO**

Deactivate the compiler problem determination facilities while compiling.

***YES**

Activate the compiler problem determination facilities while compiling.

Examples

The following command compiles a program named PAYROL.

```
CRTPLIPGM PAYROL TEXT ('Payroll Program')
```

The source program is in the default source file QPLISRC, in a member named PAYROL. A compiler listing is generated. The program is run under the *USER user profile, and can be run by any system user.

The following command creates a PL/I program named PARTS.

```
CRTPLIPGM PGM(PARTS) +  
  SRCFILE (MYLIB/PARTDATA) +  
  OPTION (*XREF *OPT) AUT (*EXCLUDE) +  
  TEXT ('This program displays all parts data')
```

The program object is stored in the library pointed to by *CURLIB. The source program is in the PARTS member of the source file PARTDATA in the library MYLIB. A compiler listing, cross-reference listing, and compiler-option list is generated. This program, which cannot be used by the public, can be run by the owner or another user that the owner has explicitly authorized by name with the CL command GRTOBJAUT (Grant Object Authority).

USING COMPILER DIRECTIVES

The *DIAGNOSE Option of the GENOPT Parameter

The *DIAGNOSE option provides the following program-checking functions at run time:

- Checking of substring range for non-adjustable strings is done automatically. Because of the many ways a string can be referenced, not all substring range violations will be detected. When a string range exception is raised by the machine, PL/I raises the ERROR condition.
- The attributes of EXTERNAL variables are matched to their external descriptions. If the attributes do not match, a diagnostic message is issued.
- All PL/I runtime informational messages which normally go to the program log are also written to the PL/I file SYSPRINT. Therefore, both user-written debug information and compiler-generated information will be intermixed in the order in which they occur.
- If you specify *DIAGNOSE, OS/400, MCH, and PL/I messages will remain on the program message queue.
- The STRINGSIZE condition informational message is sent when the STRINGSIZE condition is raised.

Using Compiler Directives

Compiler directives are statements that direct the operation of the compiler. They always begin with the percent symbol (%). They are:

```
%INCLUDE  
%PAGE  
%PROCESS (*PROCESS)  
%SKIP.
```

The %PAGE and %SKIP directives are **listing control directives**.

The %PROCESS statement is used for multiple compilation.

The %INCLUDE statement has two different uses:

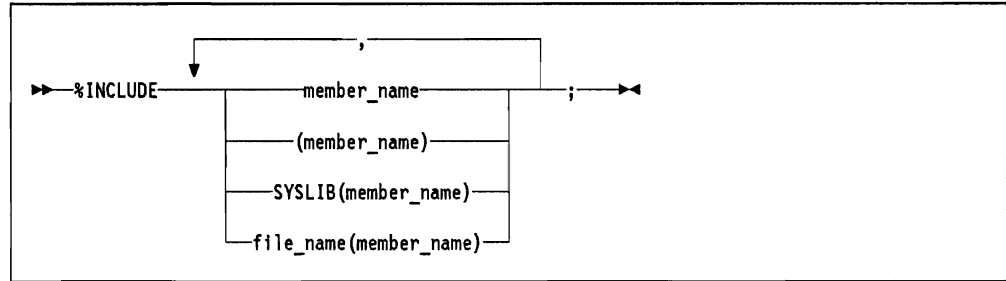
- Copying external text into the source program.
- Copying Data Description Specifications (DDS) for externally described files into the source program.

Using the %INCLUDE Directive

The %INCLUDE directive can be used to copy external text into the source program and copy DDS for externally described files into the source program.

Including External Text

The %INCLUDE directive, when used with the following syntax, includes external text into the source program.



member_name

An identifier of up to ten characters. The name must be unique within one file.

- The `member_name` specifies the name of the file member included into the source program.
- If the `member_name` appears in more than one file in your library list, and you do not specify the `file_name` or `SYSLIB`, the member used is from the first file with a member of that name found on the library list.
- You can specify up to twenty `member_names` in any `%INCLUDE` statement.

SYSLIB

This name is included for compatibility. If you enter `SYSLIB` instead of a `file_name`, the name used is the one specified on the `INCFILE` parameter of the Create PL/I Program (CRTPLIPGM) command. Refer to “Compiling Your Source Program Using the CRTPLIPGM Command” on page 2-5 for a description of the parameters of the CRTPLIPGM command.

file_name

An identifier of up to ten characters. The file is located by using the `*LIBL` search list in effect at compile time. The file name can begin with and contain numeric characters and periods. The valid characters are: A-Z, 0-9, #, \$, @, _ . You cannot name your file `SYSLIB`.

`SYSLIB` and parentheses on either side of a member name are supported for compatibility with other implementations of PL/I.

Included text can contain `%INCLUDE` directives, nested to a maximum depth of 64 levels.

The included text can be parts of statements. This provides an efficient way of creating identical declarations for different structure variables. For example:

```
DECLARE 1 A,
%INCLUDE X;
DECLARE 1 Y,
%INCLUDE X;
```

where X contains:

```
2 B BINARY,
2 C FIXED;
```

results in:

USING COMPILER DIRECTIVES

```
DECLARE 1 A,  
        2 B BINARY,  
        2 C FIXED;  
DECLARE 1 Y,  
        2 B BINARY,  
        2 C FIXED;
```

You can include text that consists of a constant, an identifier, a delimiter, or one external procedure. You cannot include any of the following:

- Parts of constants
- Parts of identifiers
- Parts of delimiters
- More than one external procedure
- The %PROCESS directive.

IBM Extension

Including DDS

For a discussion of how to use the %INCLUDE directive to copy DDS, see “Using the %INCLUDE Directive for External File Descriptions” on page 8-73.

End of IBM Extension

IBM Extension

Using the %PAGE Directive

The %PAGE directive controls the source program listing when the program is compiled. The text following a %PAGE directive is printed in the source program listing starting on the first line of the next page. The %PAGE directive does not appear in the source program compile listing.

```
▶▶%PAGE;▶▶
```

The %PAGE directive must be the only text on a line. No label prefix or comment may be specified on this line.

End of IBM Extension

Using the %PROCESS Directive

The %PROCESS directive supports batched compilation. The syntax is:

```
▶▶%  
  *  
  PROCESS  
  compiler_options  
  ;▶▶
```

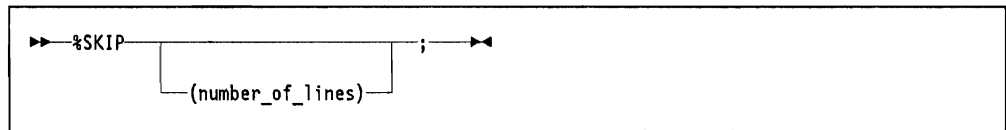
The % or * must be coded in column 1. PROCESS must be coded in columns 2 through 8. The text and semicolon must be coded in columns 9 through 72 in the first line and, if necessary, in columns 1 through 72 in subsequent lines. Text appearing after the semicolon on the same line is ignored.

The * is provided for compatibility with other compilers. The text or any options specified with this directive are ignored. You cannot copy the %PROCESS directive into your program with the %INCLUDE directive. If the %PROCESS directive precedes the first program in the source file, it must be the first record in the file.

IBM Extension

Using the %SKIP Directive

The %SKIP directive controls the source program listing when the program is compiled. The specified number of lines following a %SKIP directive in the program listing are left blank. The %SKIP directive does not appear in the source program listing.



number_of_lines

An integer constant in the range 1 through 99. It specifies the number of lines left blank. If you omit this, 1 is assumed.

The %SKIP directive must be the only text on a line. No label prefix or comment may be specified on this line.

If number_of_lines is greater than the number of lines remaining on the page, the rest of the page is skipped and printing continues at the top of the new page. In this case, the %SKIP directive is equivalent to a %PAGE directive.

End of IBM Extension

Compiler Output

In this example, the CRTPLIPGM command was entered as follows:

```

CRTPLIPGM QTEMP/LP1413 PLIST/PLISRC +
  OPTION(*XREF *OPT *AGR *ATR) +
  GENOPT(*LIST *XREF *PATCH *DUMP *ATR *DIAGNOSE)
  
```

The program source listing can be seen in Figure 8-3 on page 8-5.

The components of the listing that are produced by the various options of the GENOPT parameter document the translation of the program into machine lan-

COMPILER OUTPUT

guage. They are discussed at "Examples of Using Compiler Debugging Options" on page A-10.

```

5728PL1 R01 M00 880715          PL/I Compilation Options          PLITST/LP1413          11/30/88 15:51:24          Page 2
                                PL/I Compiler Options in Effect      1
                                AGGREGATE
                                NOATTRIBUTES
                                NOSECLVL
                                FLAG( 0)
                                GENERATE
                                GENLVL(15)
                                MARGINS(2,72)
                                OPTIONS
                                SOURCE
                                XREF
                                PL/I Generation Options in Effect      2
                                NOATTRIBUTES
                                NODIAGNOSE
                                NODUMP
                                LIST
                                NOPATCH
                                XREF
                                NOOPTIMIZE
  
```

A listing of the options (specified explicitly or by default) which were in effect when the program was compiled is produced when you specify the *OPT option in the OPTION parameter of the CL command CRTPLIPGM.

- 1 The options which were in effect for the compilation process.
- 2 Options specifying the debugging aids the compiler can generate. Many of them are discussed at "Examples of Using Compiler Debugging Options" on page A-10.

```

5728PL1 R01 M00 880715          Attribute/Cross Reference Table      PLITST/LP1413          12/10/88 11:47:31          Page 4
                                LP1413: PROCEDURE;                      PUB00160
STMT.SUBS  Identifiers          Attributes and References
4 1 BIT_FLAGS          4 STATIC /* STRUCTURE */
2 IN_FILE              5,7,9,13,15
2.1 2 IND_FILE         FILE RECORD INPUT SEQUENTIAL CONSECUTIVE BUFSIZE(38)
3.3 INDEX_BAL 3 /* In: INDEX_RECORD */ AUTOMATIC UNALIGNED PICTURE FIXED(8,2)
3.1 INDEX_KEY /* In: INDEX_RECORD */ AUTOMATIC UNALIGNED CHARACTER(10)
3.2 INDEX_NAME /* In: INDEX_RECORD */ AUTOMATIC UNALIGNED CHARACTER(20)
3 INDEX_RECORD AUTOMATIC /* STRUCTURE */
3.7 INPUT_BAL /* In: INPUT_RECORD */ AUTOMATIC UNALIGNED PICTURE FIXED(8,2)
3.5 INPUT_KEY /* In: INPUT_RECORD */ AUTOMATIC UNALIGNED CHARACTER(10)
3.6 INPUT_NAME /* In: INPUT_RECORD */ AUTOMATIC UNALIGNED CHARACTER(20)
3.4 INPUT_RECORD AUTOMATIC /* STRUCTURE */
1 LP1413 9,11,13
4.1 MORE_RECORDS EXTERNAL ENTRY
/* In: BIT_FLAGS */ STATIC ALIGNED BIT(1)
5
4.2 NO /* In: BIT_FLAGS */ STATIC ALIGNED BIT(1) INITIAL
5
4.3 YES /* In: BIT_FLAGS */ STATIC ALIGNED BIT(1) INITIAL
6
  
```

To produce a cross-reference listing of all of the variables in your program, specify the *XREF option in the OPTION parameter of the CL CRTPLIPGM command.

- 1** Statement numbers for the declarations of each of the variables.
- 2** When a data item is declared as part of a multiple declaration, both the statement and substatement where the item is declared are specified.
- 3** The identifiers used in the program (including names of procedures, files, and structures).
- 4** Attributes of each of the items and the numbers of the statements in which each item is referenced. Since the CRTPLIPGM command specifies *ATR as an option of the OPTION parameter, the attributes of each item declared in the program are also listed. If you specify both *XREF and *ATR, the compiler produces a combined cross-reference and attribute table; but you can specify one or the other and obtain a list of statements declaring and referencing the variables, or a list of the attributes of each item declared.

5728PL1 R01 M00 880715 PL/I Aggregate Length Table PLITST/LP1413 11/30/88 15:51:24 Page 6

LP1413: PROCEDURE; PUB00160

| STMT.SUBS | Identifier | LVL | DIMS | Offset From Major | Element Length | Aggregate Length |
|---------------------------------------|--------------------|----------|------------|-------------------|----------------|------------------|
| 4 1 | BIT_FLAGS 3 | 4 | 1 5 | | 3 | |
| | MORE_RECORDS | | 2 | | 0(1) | |
| | ---padding-- | | 2 | 0(1) | 0(7) | |
| | NO | | 2 | 1 | 0(1) | |
| | --padding-- | | 2 | 1(1) | 0(7) | |
| | YES | | 2 | 2 | 0(1) | |
| | --padding-- | | 2 | 2(1) | 0(7) | |
| 3 | INDEX_RECORD | | 1 | | 38 | 8 |
| | INDEX_KEY | | 2 | | 10 | |
| | INDEX_NAME | | 2 | 10 6 | 20 | |
| | INDEX_BAL | | 2 | 30 | 8 | |
| 3.4 2 | INPUT_RECORD | | 1 | | 38 | 38 |
| | INPUT_KEY | | 2 | | 10 7 | |
| | INPUT_NAME | | 2 | 10 | 20 | |
| | INPUT_BAL | | 2 | 30 | 8 | |
| Sum of Constant Lengths - 79 9 | | | | | | |

The aggregate length table is produced when you specify the *AGR option in the OPTION parameter of the CRTPLIPGM command.

- 1** The statement in which the aggregate was declared.
- 2** When an aggregate is declared as part of a multiple declaration, both the statement and substatement for the declaration are specified.
- 3** The names of the aggregates and their components.
- 4** The level number of each identifier.
- 5** For arrays, the number of dimensions in the array is listed.
- 6** The offset of each element from the first byte of storage occupied by the aggregate.
- 7** The length of each element, and each aggregate's total length.
- 8** The length of each aggregate.
- 9** The total number of bytes occupied by the aggregates.

Running the Program

The most common ways to run a PL/I program are:

- Using the CL command CALL as part of a batch job, entered interactively by the work station user, or included in a CL program.
- Using the PL/I statement CALL in a PL/I program (see “CALL Statement” on page 14-7).
- Using the other AS/400 language call statements
- Using a menu, from which the user can choose an option that calls the program.
- Creating your own command (see the *CL Programmer's Guide* for information on creating your own command.)

For more information on how to call a compiled program, see the *CL Programmer's Guide*.

Interrupting or Ending the Running of a Compiled Program

You can interrupt or end the running of a compiled PL/I program as follows.

If you are running the program from a batch job, you should issue the CL command ENDBCHJOB (End Batch Job). For information on the ENDBCHJOB command, see the *Programming: Control Language Reference*.

If you are running the program interactively, press the Sys Req key to interrupt. Then press Enter to get the System Request Menu. You then have a choice of various options from the System Request Menu. For example, if you want to end processing of the program, enter option 2. If you want to resume the processing of the program, either press the F3 key, or press Enter with the options field left blank.

Abnormal Program Ending

If the processing of a PL/I run unit ends abnormally, escape message PLI9001 or PLI9002 or PLI9003 is sent to the program that called the first procedure in the run unit. This results in a function check if the escape message is not monitored. In a CL program, you can monitor these messages with the CL command MONMSG (Monitor Message). See the *Programming: Control Language Reference* for more information.

A program that causes an error that cannot be handled through the normal flow of control will set the return code to 4. If the program processes a SIGNAL statement for the ERROR condition, the return code is set to 3. The processing of a STOP statement or a call to PLIDUMP with the stop option (S) will set the return code to 2. For information on return codes, see the entries in the *Programming: Control Language Reference* on the CL commands RTVJOBA (Retrieve Job Attributes) and WRKJOB (Work With Job).

Interlanguage Calls

The AS/400 System allows you to call programs written in different languages. The techniques used for transferring between programs and passing parameters are similar to those used for communicating between different programs written in PL/I. There are two cases to consider:

- When your PL/I program calls a program written in another language
- When a program written in another language calls your PL/I program.

Calling a Non-PL/I Program

In a calling PL/I program, you must code an ENTRY declaration for the program you will link to. For example:

```
DECLARE COBOLPGM ENTRY
      CHARACTER (8),
      FIXED DECIMAL (4,1),
      FIXED DECIMAL (3)),
      OPTIONS (ASSEMBLER);
```

The OPTIONS (ASSEMBLER) attribute tells the compiler that the interface with the called program will be at the machine interface level: that is, that PL/I will pass parameters directly to the program instead of using PL/I control blocks.

Arrays are not supported as parameters between PL/I and BASIC, because PL/I does not build the array descriptors which BASIC requires for arrays.

After the ENTRY attribute, you should list '+' attributes (not the variable names) of the parameters passed to the called program.

When you call a non-PL/I program, you list the variables that you are using as parameters. These variables must be declared with exactly the same attributes as those you have listed in the ENTRY declaration. For the ENTRY declaration given above, you may declare the parameter variables as follows:

```
DECLARE ITEM1 CHARACTER (8),
      ITEM2 FIXED DECIMAL (4,1),
      RESULT FIXED DECIMAL (3);
```

You would then initialize the variables for which you wish to pass a value to the program you are calling:

```
GET FILE (SYSIN) EDIT (ITEM1,ITEM2)(A(8),F(4,1));
```

The actual CALL statement has the same format as for the calling of a PL/I procedure:

```
CALL COBOLPGM (ITEM1,ITEM2,RESULT);
```

Control is passed to COBOLPGM, which could use ITEM1 and ITEM2 as input data and place a value in RESULT that will be returned to the calling PL/I program. When the called program finishes running, control is returned to the statement following the call statement.

INTERLANGUAGE CALLS

If a non-PL/I program ends abnormally, the ERROR condition is raised in the calling program. You can use an on-unit to take appropriate action.

Calling a PL/I Program from a Non-PL/I Program

In a PL/I program called by a program that is coded in another language, you must list the parameters in the PROCEDURE statement at the start of the program:

```
SUBPGM: PROCEDURE (INTEGER1,INTEGER2,CHAR1);
```

The parameter variables must be declared inside the PL/I program:

```
DECLARE INTEGER1 FIXED BINARY (15),
        INTEGER2 FIXED DECIMAL (7),
        CHAR1 CHARACTER (8);
```

The attributes of the parameter variables declared in the PL/I program must exactly match the attributes in the calling program. You cannot use asterisks or variables to indicate the length of a character or bit scalar data item, or the bounds of an array, as in:

```
DECLARE CHARITEM CHARACTER (*),
        ARRAY1(INDEX1) FIXED DECIMAL (7),
        BITARRAY(*) BIT (*) ALIGNED;
```

The necessary PL/I control blocks which furnish these values when the program is called are available only when the calling program is written in PL/I.

When a floating-point value is passed to a PL/I program from CL, the variable that the data is placed in must have the UNALIGNED attribute in the PL/I program.

The following tables show you how to code matching data types in PL/I and the other languages available on the AS/400 System.

| Data type | PL/I | RPG | |
|-------------------|--|---------------------------|---|
| Packed Decimal | FIXED DECIMAL (p,q) | Columns | Code |
| | Where: | | |
| | p = total number of digits and $1 \leq p \leq 15$. | 6 | I |
| | q = number of digits to the right of the decimal point and $1 \leq q \leq 15$. | 43 44-47 | P a, where $b-a+1 =$ $(p+1)/2$ and $1 \leq p \leq 15$ |
| | p = greater than or equal to q | 48-51 52 | b q, where q is the number of decimal digits |
| | | 53-58 | name of the packed field |

Figure 2-7 (Part 1 of 3). Matching PL/I Attributes in RPG

| Data type | PL/I | RPG | |
|---------------|--|---|--|
| Zoned Decimal | <p>PICTURE 'p'</p> <p>Where:</p> <p>p is the number of 9s</p> <p>- or -</p> <p>PICTURE 'pVq'</p> <p>Where:</p> <p>p is the number of 9s to the left of the V and $1 \leq p \leq 15$</p> <p>V is the implied decimal point</p> <p>q is the number of 9s to the right of the V and $1 \leq q \leq 15$</p> <p>p is greater than or equal to q</p> | <p>Columns</p> <p>6</p> <p>43</p> <p>44-47</p> <p>48-51</p> <p>52</p> <p>53-58</p> | <p>Code</p> <p>I</p> <p>blank</p> <p>a, where a is the starting position of the field</p> <p>b, where b is the end position of the field</p> <p>q, where q is the number of decimal digits and $0 \leq q \leq 9$</p> <p>name of the zoned subfield</p> |
| Fixed Binary | <p>FIXED BINARY (p)</p> <p>Where:</p> <p>p = total number of binary digits and $1 \leq p \leq 31$</p> | <p>Columns</p> <p>6</p> <p>43</p> <p>44-47</p> <p>48-51</p> <p>52</p> <p>53-58</p> | <p>Code</p> <p>I</p> <p>B</p> <p>a, where a is the starting position of the field</p> <p>b, where b is the end position of the field</p> <p>q, where q is the number of decimal digits and $0 \leq q \leq 9$</p> <p>name of the binary subfield</p> |
| Float Decimal | <p>FLOAT DECIMAL (p)</p> <p>Where:</p> <p>p = total number of digits and $1 \leq p \leq 16$</p> | Not supported. | |
| Float Binary | <p>FLOAT BINARY (p)</p> <p>Where:</p> <p>p = total number of digits and $1 \leq p \leq 53$</p> | Not supported. | |

Figure 2-7 (Part 2 of 3). Matching PL/I Attributes in RPG

INTERLANGUAGE CALLS

| Data type | PL/I | RPG | | | | | | |
|--------------------------|--|--|----------------|-------------|--------------|------|--------------|------------------------------|
| Bit | BIT (w) Where: w = total number of bits and $1 \leq w \leq 32767$ | The use of bit strings is not supported. A bit string can be passed as a character string in which the length of the character string in bytes is equal to $(w + 7)/8$. The receiving program must define the bits in the bytes being passed. | | | | | | |
| Character | CHARACTER (w) Where: w = total number of characters and $1 \leq w \leq 32767$ | <table border="0"> <tr> <td>Columns</td> <td>Code</td> </tr> <tr> <td>28-32</td> <td>PARM</td> </tr> <tr> <td>49-51</td> <td>w, where $1 \leq w \leq 999$</td> </tr> </table> | Columns | Code | 28-32 | PARM | 49-51 | w, where $1 \leq w \leq 999$ |
| Columns | Code | | | | | | | |
| 28-32 | PARM | | | | | | | |
| 49-51 | w, where $1 \leq w \leq 999$ | | | | | | | |
| Varying Length Character | CHARACTER (w) VARYING Where: w = total number of characters and $1 \leq w \leq 32765$ | Not supported. | | | | | | |

Figure 2-7 (Part 3 of 3). Matching PL/I Attributes in RPG

| Data type | COBOL | BASIC | CL |
|----------------|--|---|--|
| Packed Decimal | PIC S9(p)V9(q) USAGE COMP-3 Where: $1 \leq p \leq 15$ $1 \leq q \leq 15$ - or - PIC S9(p) USAGE COMP-3 Where: $1 \leq p \leq 15$ $q = 0$ | DECLARE PROGRAM ... PD p,q Where: $1 \leq p \leq 15$ $1 \leq q \leq 15$ - or - DECLARE PROGRAM ... PD p Where: $1 \leq p \leq 15$ $q = 0$ | TYPE(*DEC) LEN(p q) Where: $p = 15$ $q = 5$ |

Figure 2-8 (Part 1 of 3). Matching PL/I Attributes in COBOL, BASIC, and CL

| Data type | COBOL | BASIC | CL |
|------------------|---|--|---|
| Zoned Decimal | <p>PIC S9(p)V9(q) USAGE DISPLAY</p> <p>Where: p = 15 1 ≤ q ≤ 15</p> <p>- or -</p> <p>PIC S9(p) USAGE DISPLAY</p> <p>Where: 1 ≤ p ≤ 15</p> | <p>DECLARE PROGRAM ... ZD p,q</p> <p>Where: 1 ≤ p ≤ 15 1 ≤ q ≤ 15</p> <p>- or -</p> <p>DECLARE PROGRAM ... ZD p</p> <p>Where: 1 ≤ p ≤ 15 q = 0</p> | Not supported. |
| Fixed Binary | <p>PIC S9(4) USAGE COMP-4</p> <p>Where: 1 ≤ p ≤ 15</p> <p>- or -</p> <p>PIC S9(9) USAGE COMP-4</p> <p>Where: 1 ≤ p ≤ 15</p> | <p>INTEGER</p> <p>Where: 1 ≤ p ≤ 15</p> <p>- or -</p> <p>DECLARE PROGRAM ... B 2</p> <p>Where: 1 ≤ p ≤ 15</p> | Not supported. |
| Float Decimal | Not supported. | <p>DECIMAL</p> <p>Where: 1 ≤ p ≤ 6</p> <p>- or -</p> <p>DECLARE PROGRAM ... S</p> <p>Where: 1 ≤ p ≤ 7</p> | <p>A floating-point literal with double precision</p> <p>Where: 1 ≤ p ≤ 6</p> |

Figure 2-8 (Part 2 of 3). Matching PL/I Attributes in COBOL, BASIC, and CL

INTERLANGUAGE CALLS

| Data type | COBOL | BASIC | CL |
|--------------------------------|--|--|--|
| Float Binary | Not supported. | DECIMAL Where: $p > 24$ - or - DECLARE PROGRAM ... S Where: $1 \leq p \leq 24$ | A floating-point literal with double precision Where: $1 \leq p \leq 24$ |
| Bit | The use of bit strings is not supported. A bit string can be passed as a character string in which the length of the character string in bytes is equal to $(w + 7)/8$. The receiving program must define the bits in the bytes being passed. | The use of bit strings is not supported. A bit string can be passed as a character string in which the length of the character string in bytes is equal to $(w + 7)/8$. The receiving program must define the bits in the bytes being passed. | The use of bit strings is not supported. A bit string can be passed as a character string in which the length of the character string in bytes is equal to $(w + 7)/8$. The receiving program must define the bits in the bytes being passed. |
| Character | PICTURE X(w) Where: $1 \leq w \leq 32\ 767$ | DECLARE PROGRAM ... C w Where: $1 \leq w \leq 255$ | TYPE(+CHAR) LEN(w) Where: $1 \leq w \leq 2\ 000$ |
| Varying Length Character | 01 name-1. 02 name-2 PIC S9999 COMP-4. 02 name-3 PIC X(w) OCCURS DEPENDING ON name-2. Where: $1 \leq w \leq 9\ 999$ | DECLARE PROGRAM ... V Where: $1 \leq w \leq 255$ | Not supported. |

Figure 2-8 (Part 3 of 3). Matching PL/I Attributes in COBOL, BASIC, and CL

Chapter 3. Testing and Debugging PL/I Programs

Both OS/400 and PL/I offer features that you can use to test and debug your PL/I programs.

OS/400 provides:

- Test library
- Breakpoints
- Traces
- A debug feature.

PL/I provides:

- PLIDUMP
- An error dump option screen
- PLIIOFDB
- PLIOPNFDB
- ON conditions.

Note: Some of these PL/I features may use OS/400 functions to provide input.

The OS/400 features let you test programs while protecting your production files, and let you observe and debug operations as a program runs. No special source code is required to use the OS/400 features.

The PL/I features can be used independently of the OS/400 functions or in combination with them to:

- Debug a program
- Produce a formatted dump of the contents of fields, data structures, arrays, and tables.

Source code in the form of compiler directives is required to use the PL/I debugging features and formatted dump.

Using, Displaying, and Printing Messages

Using Messages

This manual refers to messages you receive from the compiler. These messages are displayed on your screen or printed on your compiler listing. There are no message manuals for this product.

MESSAGES

Each compiler message contains a minimum of three parts as illustrated in the following screen example:

```

A MSGID: PLC2188 Severity: 20
B Message . . . . : An unexpected continuation was found. An
    end of statement is assumed before 'PUT SKIP EDIT ( A'.
C Cause . . . . : This text cannot be interpreted as a
    continuation of the statement. A delimiter, such as an
    operator in the expression or a semicolon may be missing.
    The compiler ignores the text up to the next semicolon.
Recovery . . . : Check for a missing delimiter.
```

A A number indicating the severity of the error.

Severity Meaning

- 00 An *informational message* displayed during entering, compiling, and running: This level is used to convey information to the user. No error has been detected and no corrective action is necessary.
- 10 A *warning message* displayed during entering, compiling and running: This level indicates that an error was detected but is not severe enough to interfere with the running of the program. The results of the operation are assumed successful.
- 20 An *error message* displayed during compiling: This level indicates that an error was made, but the compiler is taking a recovery that might yield the desired code. The program may not work as the author intended.
- 30 A *severe error message* displayed during compiling: This level indicates that an error too severe for automatic recovery was detected. Compilation is completed, but running the program cannot be attempted.
- 40 An *abnormal end of program* or *function message* displayed during running: This level indicates an error that forces cancellation of processing. The operation may have ended because it was unable to handle valid data, or possibly because the user cancelled it.
- 50 An *abnormal end of job message* displayed during running: This level indicates an error that forces cancellation of job. The job may have ended because a function failed to perform as required, or possibly because the user cancelled it.
- 99 A *user action taken during running*: This level indicates that some manual action is required of the operator, such as entering a reply, changing diskettes, or changing printer forms.

B The text you see online or on a listing. This text is a brief, generally one sentence, description of the problem.

C The text you see online when you press F4 from the screen with the first-level text. This text will be printed on your listing if you specify *SECLVL in your compile-time options. The IBM-supplied default for this option is *NOSECLVL. This

text contains an expanded description of the message (Cause) and a section detailing the correct user response (Recovery).

Displaying and Printing Messages

To display or print a particular message or messages, use the DSPMSGF or DSPMSGD commands. These commands are described in the *Programming: Control Language Reference*.

Note: If you have any comments or suggestions concerning the messages, please use the Reader Comment Form included with this manual and send them to us.

Using a Test Library

Programs that run in a normal operating environment can read, update, and write records in both test and production libraries. Programs that run in a testing environment can also read, update, and write records in both test and production libraries. However, to prevent data base files in production libraries from being accidentally changed, you can use UPDPROD(*NO) in the CL command STRDBG (Start Debug) or in the CL command CHGDBG (Change Debug). See the *Programming: Control Language Programmer's Guide* and the *Programming: Control Language Reference* for more information.

On the AS/400 System, you can copy production files into a test library or you can create special files for testing in the test library. A production file and its test copy can have the same name if they are in different libraries. You can then use the same file name in the program for either testing or normal processing.

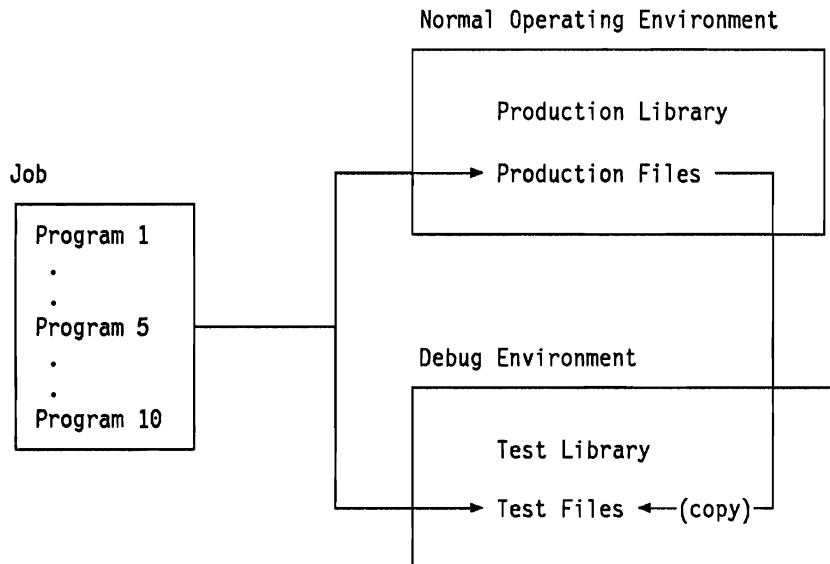


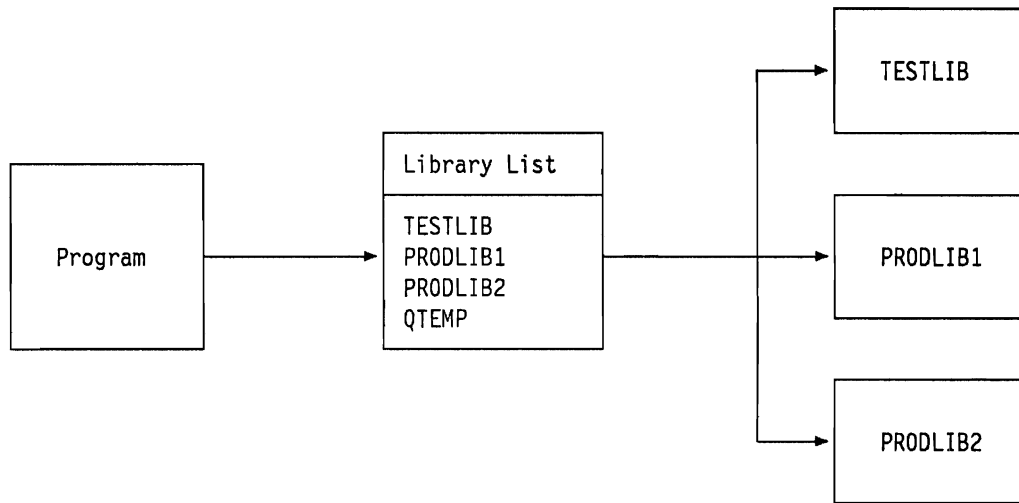
Figure 3-1. Using a Test Library

For testing, you must put the test library name ahead of the production library name in the library list for the job that contains the program tested. For normal

USING TEST LIBRARIES

processing, the test library should not be named in the library list for the job, as shown in the following diagram.

Debugging



Normal Operating Environment

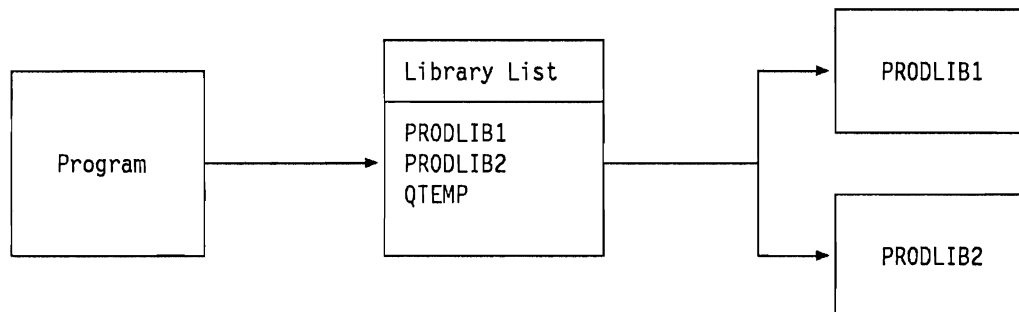


Figure 3-2. Using a Library List

No special statements for testing are necessary within the program being tested. The program can be run normally without any changes. All debug functions are given in the job that contains the program instead of in the program. However, you can include statements such as `CALL PLIDUMP` in your program if you need them.

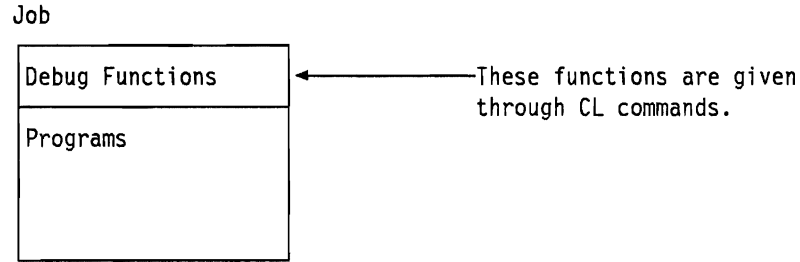


Figure 3-3. Using Debug Functions

Debug functions apply only to the job in which they are given. A program can be used at the same time in two jobs: one job that is in a testing environment, and another job that is in a normal operating environment.

Using Breakpoints

A breakpoint is a point in your program where you want the program to stop running and wait. You can use any of the following as a breakpoint:

- A statement number from the compiled program source listing.
- A machine interface (MI) instruction number from an IRP program listing.

You cannot use SEU2 source sequence numbers or labels and procedure names from the program.

When a breakpoint statement is about to be processed in an interactive job, the system displays the breakpoint at which the program is stopped. The values of the program variables you have asked for on the ADDBKP command, if any, are displayed. After this information is displayed, press F10 to get the command entry screen, from which you can enter CL commands to ask for other functions (such as displaying or changing a variable value, adding a breakpoint, or adding a trace), or press Enter to continue processing, or press F3 to cancel the program or function being processed.

For a batch job, a breakpoint program can be called when a breakpoint is reached in the program being tested. The breakpoint information is passed to the breakpoint program. For a description of the actual parameters passed, see the description of the BKPPGM parameter of the CL command ADDBKP in the *Programming: Control Language Reference*.

Example of Using Breakpoints

The following CL program calls the program shown in Figure 8-5 on page 8-10, adds breakpoints, and displays the values on the screen.

USING BREAKPOINTS

```
5728PW1 R01M00 880715          SEU SOURCE LISTING          11/30/88 09:51:06          PAGE 1
SOURCE FILE . . . . . PLITST/CL
MEMBER . . . . . BREAKPT
SEQNBR*...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 0
100 PGM
200 ENTDBG PGM(LP1414) UPDPROD(*YES)
300 ADDBKP STMT(18)
400 ADDBKP STMT(20) PGMVAR((INPUT_KEY))
500 CALL PGM(LP1414)
600 EHDDBG
700 ENDPGM
06/10/83
01/09/85
03/01/84
03/01/84
03/01/84
06/10/83
```

Figure 3-4 (Part 1 of 2). CL Program and Display for Breakpoints

```

                                Display Program Variables
Program . . . . . : LP1414
Invocation level . . . . . : 1
Start position . . . . . : 1
Format . . . . . : *CHAR
Length . . . . . : *DCL
Variable . . . . . : 02 INPUT_KEY
  Type . . . . . : CHARACTER
  Length . . . . . : 20
  *...+...1...+...2...+...3...+...4...+...5
  '1111  '

Press Enter to continue.
F3=Exit  F12=Previous
```

Figure 3-4 (Part 2 of 2). CL Program and Display for Breakpoints

Considerations for Using Breakpoints

You should be aware of the following before you use breakpoints:

- If a breakpoint is bypassed by a PL/I statement, such as GOTO, that breakpoint is ignored.
- When a breakpoint is added for a statement, the program stops just before the statement is processed.
- Breakpoint functions are specified through CL commands.

These functions include adding breakpoints to programs, displaying breakpoint information, removing breakpoints from programs, and continuing to run a program after a breakpoint display is shown. See the *Programming: Control Language Reference* for information on these commands, and the *Programming: Control Language Programmer's Guide* for more information about breakpoints.

Using a Trace

A trace is a record of some or all of the statements in a compiled program that were processed and the values of any variables that were specified on the CL command ADDTRC. A trace is different than a breakpoint in that you are not given control during the trace.

The system records the traced statements that were processed. You must ask for a display of the traced information using the CL command DSPTRCDTA. The display shows the sequence in which the statements were processed and the values of variables you specified.

You enter the statements that the system should trace. You can also specify that variables be recorded or displayed before each traced statement is processed, or only when the value of some traced variable changes from the last time a traced statement was processed.

You can request a trace of one statement in a program, a group of statements in a program, or all the statements in a program.

Example of Using a Trace

The following CL program adds a trace requests, calls the program shown in Figure 8-5 on page 8-10, and displays the trace data.

```

5728PW1 R01M00 880715          SEU SOURCE LISTING          11/30/88 09:54:53          PAGE 1
SOURCE FILE . . . . . PLITST/CL
MEMBER . . . . . TRACE
SEQNBR*...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 0
100 PGM
200 ENTDBG PGM(LP1414) UPDPROD(*YES)
300 ADDTRC STMT((19 23)) PGMVAR((INPUT_KEY)) OUTVAR(*CHG)
400 CALL PGM(LP1414)
500 DSPTRCDTA CLEAR(*YES)
600 ENDDBG
700 ENDPGM
                                06/18/83
                                01/09/85
                                03/01/84
                                03/01/84
                                01/09/85
                                06/18/83

```

Figure 3-5 (Part 1 of 2). CL Program for Requesting a Trace and Displaying Trace Data

USING TRACES

```
1/07/88 18:49:50      TRACE DATA DISPLAY
Stmt/Inst:  19          Pgm:  LP1414      Inv lvl:  1          1
Start pos:   1          Len:  *DCL        Format:  *CHAR
Variable:   02 INPUT_KEY
Type:       CHARACTER      Length:   10
*...+...1...+...2...+...3...+...4...+...5
'11111'
Stmt/Inst:  23          Pgm:  LP1414      Inv lvl:  1          2
Stmt/Inst:  20          Pgm:  LP1414      Inv lvl:  1          3
Stmt/Inst:  20          Pgm:  LP1414      Inv lvl:  1          4
Stmt/Inst:  22          Pgm:  LP1414      Inv lvl:  1          5
Stmt/Inst:  23          Pgm:  LP1414      Inv lvl:  1          6
Start pos:   1          Len:  *DCL        Format:  *CHAR
*Variable:  02 INPUT_KEY
Type:       CHARACTER      Length:   10
*...+...1...+...2...+...3...+...4...+...5
'222211111'
Stmt/Inst:  20          Pgm:  LP1414      Inv lvl:  1          7
Stmt/Inst:  21          Pgm:  LP1414      Inv lvl:  1          8
Stmt/Inst:  22          Pgm:  LP1414      Inv lvl:  1          9
Stmt/Inst:  23          Pgm:  LP1414      Inv lvl:  1         10
Start pos:   1          Len:  *DCL        Format:  *CHAR
*Variable:  02 INPUT_KEY
```

Figure 3-5 (Part 2 of 2). CL Program for Requesting a Trace and Displaying Trace Data

Considerations When Using a Trace

You should be aware of the following before you use traces with PL/I programs:

- If a group of PL/I statements is bypassed, they are not included in the trace. The case is similar with breakpoints (see “Considerations for Using Breakpoints” on page 3-6).
- Trace functions are given by CL commands in the job that contains the traced program.

These functions include adding trace requests to a program, removing trace requests from a program, removing data collected from previous traces, displaying trace information, and displaying the traces that have been entered for a program.

- In addition to statement numbers, names of routines generated by PL/I can appear on the trace output STMT field.

The compiler reorganizes the source statements in your program by denesting the blocks (including procedures). The effect of denesting is illustrated below:

Program Source

| PL/I Statement Number | PL/I Source Statement |
|-----------------------------|-----------------------------|
| 1 | OUTER: PROCEDURE; . |
| 20 | INNER: PROCEDURE; . |
| 50 | ITEM1 = ITEM1 + 1; . |
| 60 | END INNER; . |
| 70 | ITEM2 = ITEM2 + 1; . |
| 100 | END OUTER; |

Sequence of MI Instructions Generated in the Source Listing When This Source is Compiled

| PL/I Statement Number | PL/I Source Statement |
|-----------------------------|-----------------------------|
| 1 | OUTER: PROCEDURE; . |
| 70 | ITEM2 = ITEM2 + 1; . |
| 100 | END OUTER; |
| 20 | INNER: PROCEDURE; . |
| 50 | ITEM1 = ITEM1 + 1; . |
| 60 | END INNER; |

This denesting of blocks has the following consequences:

- You should not specify a trace range where the starting and ending statements are contained in different blocks, because the range may not be valid or may trace a set of statements different from the one you intended. For example, the following CL command would not be valid for the example above:

```
ADDTRC STMT((50 70))
```

because statement 50 corresponds to a higher MI instruction number than statement 70 in the MI program.

- If you specify a range of statements that includes a nested block, no trace will be processed on the statements contained by the inner block. For instance,

```
ADDTRC STMT((1 100))
```

USING DEBUG

does not trace any of the statements in INNER, because the MI instructions for INNER are all beyond the MI instruction for statement 100 in the MI program.

You can always specify

```
ADDTRC STMT(*ALL)
```

to trace all statements processed in the entire program.

See the *Programming: Control Language Programmer's Guide* for more information about traces.

Using Debug

OS/400 Debug is used in a test environment that you can enter using CL commands like STRDBG (Start Debug) or CHGDBG (Change Debug). This environment allows you to use the debugging features and run the program without affecting the normal program environment. The following items should be taken into account when using debug:

- Calling Levels
- Scoping of names
- Fully qualified names
- PL/I pointers
- Floating point variables
- Changing varying length strings
- Specifying variables by ODV number
- Displaying level numbers
- References to static variables
- Determination of active blocks in a program.

PL/I Storage

PL/I variables use storage areas allocated and maintained by PL/I. The system Program Static Storage Area (PSSA) or the Program Automatic Storage Area (PASA) are not used for any PL/I variables.

Calling Levels

When you use a recursive program or procedure, you should be aware of two calling levels: the **program** calling level and the **procedure** calling level.

When a program or external procedure is called recursively, the program calling level is incremented. You can specify the program calling level on the OS/400 debug commands through the INVLVL parameter.

When an internal procedure is called recursively, the procedure calling level is incremented. You cannot specify the procedure calling level on the OS/400 debug commands. Only the last (most recent) procedure calling level is available for debugging.

Scoping of Names

In PL/I, the scope of a name is determined by the block(s) in which it is declared. A block is defined by a PROCEDURE or BEGIN statement. If a name is declared in more than one block in a program, PL/I scoping rules determine which declaration is used when you refer to the name. For more information on scoping, see "Names" on page 4-12.

OS/400 debug operates outside of the PL/I program and cannot use PL/I scoping rules to determine which declaration you are referring to. When a name is unique only because of PL/I scoping rules, debug will not be able to determine which declaration should be used (a message is issued saying the name is ambiguous).

To specify a unique reference to a name declared in more than one block, use the block number on the compiled program source listing. The block number is the highest level qualifier for a name in a PL/I program. That qualifier represents the block that the variable is declared in. When you specify the block number that the variable is declared in, OS/400 debug can determine which declaration of a name should be used.

The qualifier is of the form *BLKn, where n is a one to three digit block number.

For example, if variable K is defined in blocks 2 and 5 in your program, and you wish to display the value of K in block 5, specify a PGMVAR parameter of *BLK5.K. If block 5 is not currently active, the value of K in block 5 cannot be displayed: instead, a message is displayed that indicates that the variable is not currently active. Note that the value of K in block 5 is displayed even if block 2 is also active.

Fully Qualified Names

The test environment recognizes a concept of a fully qualified name similar to the PL/I concept. However, because every PL/I variable has a block number as the highest level qualifier, you must specify the block number qualifier (*BLKn) as a part of the fully qualified name, whenever it is necessary to specify the fully qualified name. For example, consider the following declarations:

```
DECLARE 1 SAMPLESTRUCTURE,
        5 ITEM1                FIXED BINARY (15);
DECLARE ITEM1                  CHARACTER (10);
```

A request to display or change ITEM1 is ambiguous to the OS/400 debug facility. Assuming the variables are both in block 7, you must specify

```
*BLK7.SAMPLESTRUCTURE.ITEM1
```

or

```
SAMPLESTRUCTURE.ITEM1
```

to process the variable ITEM1 that is an element of SAMPLESTRUCTURE, and

```
*BLK7.ITEM1
```

to process the scalar character variable ITEM1.

PL/I Pointers

Names that are declared with the `POINTER` attribute in the PL/I source program are called High Level Language (HLL) pointers in the test environment. These pointers are maintained at the machine level as space pointers. You can only change them in the test environment using the CL command `CHGHLLPTR` (Change High Level Language Pointer) or `CHGPTR` (Change Pointer) to contain a space pointer value or a null pointer.

The `CHGHLLPTR` command allows you to change the value of a HLL pointer. This pointer can be a pointer variable or a basing-pointer name. The value of the pointer copied can be a pointer variable or a program variable address referred to by the variable name. The reference pointer or program variable can have another HLL pointer(s) specified as its basing pointer(s).

The following CL statements illustrate the use of the command:

```
CHGHLLPTR PTR('PTR1') REFPTR('PTR2')
```

```
CHGHLLPTR PTR('PTR3(4)') ADR('VAR1(VAR2,5)' 'PTR4(3)')
```

For more information on the CL command `CHGHLLPTR`, see the *Programming: Control Language Reference*.

Floating Point Variables

In the test environment, floating point variables are displayed with the precision in which they are stored internally, and not as they are declared in your PL/I program. Short floating point variables are displayed with a precision of `BINARY FLOAT(24)` or `DECIMAL FLOAT (7)`, which requires four bytes of storage. Long floating-point variables are displayed with a precision of `BINARY FLOAT(53)` or `DECIMAL FLOAT (16)`, which requires eight bytes of storage. OS/400 debug does not use the value of the precision declared in the program before displaying or changing the value.

Changing Varying Length Strings

When you use the CL command `CHGPGMVAR` to change a varying length character string, the bytes changed must either start within the current length of the string or start at the next byte after the end of the string as defined by the current length. If the current length is negative, the length is treated as though it were 0. The length of the string is always adjusted to match the last byte changed by the command. If updating the variable exceeds the maximum length, an error message is issued and the variable is not changed.

A varying-length string can be truncated without changing the value in the part of the string that remains after truncation. To do this, specify a null string for the new value. For example:

```
CHGPGMVAR PGMVAR(VARYINGCHARSTRING) VALUE(' ') START(11)
```

truncates the current length of `VARYINGCHARSTRING` to ten characters. The byte count at the start of the string is updated to a value of ten.

Specifying Variables by ODV Number

You can display and change program data using ODV (Object Definition Table Directory Vector) numbers. These numbers are found on the program IRP listing, which is obtained by specifying GENOPT (*LIST) on the CRTPLIPGM command. A cross-reference of ODV numbers can be obtained by specifying GENOPT(*XREF) on the CRTPLIPGM command. For more information on the format of ODV numbers, refer to the *Programming: Control Language Programmer's Guide*.

If ODV numbers are used to specify the names of variables, OS/400 debug only uses the information that is defined for the variable at the machine instruction (MI) interface. The value and attributes of the variable presented to you may be very different from what would be presented if the HLL variable name was specified.

Displaying Level Numbers

Any PL/I variable declared with a structure level number is shown on the OS/400 debug display with the level number immediately preceding the variable name. The level numbers displayed by OS/400 debug start at 1 for each structure and increment by 1 for each new level in the structure. The following example shows the level numbers for a structure in a PL/I program, and the corresponding level numbers that would be displayed by OS/400 debug.

| PL/I Level Number | OS/400 Level Number |
|-----------------------------|---------------------------|
| 01 PARTS, | 01 |
| 05 ITEM, | 02 |
| 10 OLD FIXED DECIMAL (5,0), | 03 |
| 10 NEW FIXED DECIMAL (9,0), | 03 |
| 05 DESCRIPT CHAR(10); | 02 |

References to Static Variables

A variable declared in your PL/I program with the STATIC attribute can only be referenced by OS/400 debug when the program and block in which it is declared are currently active.

Determination of Active Blocks in a Program

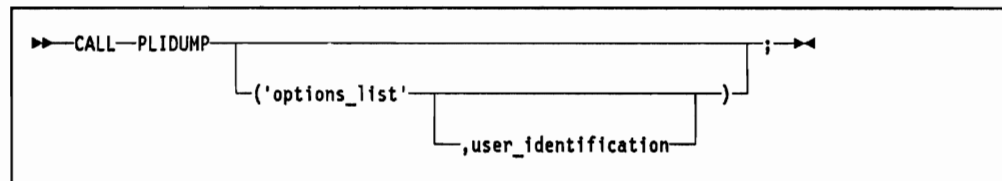
A variable can only be displayed or changed by OS/400 debug if the block that defined the variable is active. This is true regardless of the storage class of the variable. You can determine if a block is active at any point while debugging by displaying any variable that is declared in the block, or by displaying the special variable *BLKn where n is the block number that you want to check. If you receive a message that the variable is not active, the block is not currently active.

Using PLIDUMP

The PLIDUMP built-in subroutine produces a symbolic dump of the variables of the currently running program. The output of PLIDUMP appears on the system dump file QPPGMDMP.

The program variables that are dumped depend on the options you specify when you call PLIDUMP. The dump also contains:

- A list of any ONCODE, ONFILE, or ONKEY data which is relevant
- The date and time of the dump
- The statement number from which the dump was called.



options_list

A contiguous string of characters consisting of one or more of the following dump options.

T NT F NF V NV H NH S C

The dump options are described as follows. The default dump options are underlined.

- T Displays a trace of the currently active blocks in the run unit, containing the name of the blocks (if applicable), statement numbers of calling statements, and error information for on-units.
- NT No trace information is displayed.
- F Displays the symbolic attributes and record contents of the buffers of all open files.
- NF No file information is displayed.
- V Generates a dump of all AUTOMATIC and STATIC variables (with a non-zero length) for the current calling of the external procedure with their identifiers. Recursive procedures and ON units are dumped for only the most recent call.
- NV No variables are dumped.
- H Produces a hexadecimal dump of the PL/I data spaces of the environment in which the program is running. This option is provided to assist in servicing the program.
- NH No hexadecimal dump is produced.
- C Continues running the program after the dump.
- S Ends the program after the dump. When you select this option, an "Operator Requested Error Dump" will not be produced (see "Error Dump Option Screen" on page 3-16 for more information).

Options are read from left to right. Invalid options are ignored, and if contradictory options are coded, the rightmost options are used.

user_identification

A character string variable or constant chosen by the PL/I programmer. It can be of any length, but a maximum of 36 characters is printed at the head of the formatted dump. The rest is truncated. If the character string is omitted, no identification is printed.

When you are debugging, you may call PLIDUMP from an on-unit, however, it may be called from anywhere else in your program.

You can specify the C (continuation) option of PLIDUMP to get a series of dumps of storage while the program is running.

If you call PLIDUMP several times in a program, use a different user identification to identify each dump.

PLIDUMP can also be called whenever the program encounters a system error that is not handled by OS/400 or by your program (see the following section).

Example of Using PLIDUMP

The program used for the dump in Figure 3-6 is the same as that shown in Figure 8-5 on page 8-10, except that the following statement has been added to the program between statements 23 and 24:

```
CALL PLIDUMP;
```

This procedure produces the default dump as shown in Figure 3-6.

To produce the maximum amount of information, including a variable dump and hexadecimal dump, use the following statement:

```
CALL PLIDUMP('TFVHC', 'FULL PL/I PROGRAM DUMP');
```

```
18:02:28 01/07/88 PLIDUMP CALLED FROM STATEMENT 00025 PROGRAM LP1427.

CURRENT OPTIONS IN EFFECT (TFNVNHC)

TRACE
FILE
NOVARIABLES
NOHEXADECIMAL
CONTINUE
```

Figure 3-6 (Part 1 of 3). PL/I program calling PLIDUMP

ERROR DUMP OPTION SCREEN

```
TRACE OF CURRENT DSA STACK

DSA BLOCK NUMBER      - 00001
BLOCK NAME            - LP1427
FROM STATEMENT        - *EXT
PROGRAM NAME          - LP1427

END OF DSA TRACE
```

Figure 3-6 (Part 2 of 3). PL/I program calling PLIDUMP

```
SYMBOLIC DUMP OF FILE SYSPRINT

STATUS                - OPEN
ACTUAL FILE NAME      - QPRINT
SEPARATE INDICATORS  - NO
LAST OPERATION        - PUT

COMPLETED FILE ATTRIBUTES
STREAM
PRINT
OUTPUT
EXTERNAL

ENVIRONMENTAL ATTRIBUTES
CONSECUTIVE

OUTPUT BUFFER
000000  00F8F8F8  F8F8F2F2  F2F2F240  00000000  D2D9E8E3  D6D54009  C9000000  00000000  * 8888822222  KYRTON II  *
000020  00000000  00000000  00000000  00000000  F24BF4F0  00000000  00000000  00000000  *                2.40  *
000040- 00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000  *                *
000060  SAME AS ABOVE
000080  00000000
```

Figure 3-6 (Part 3 of 3). PL/I program calling PLIDUMP

Error Dump Option Screen

When your program encounters a system error that is not handled by OS/400 or by PL/I, the following display appears on your work station screen:

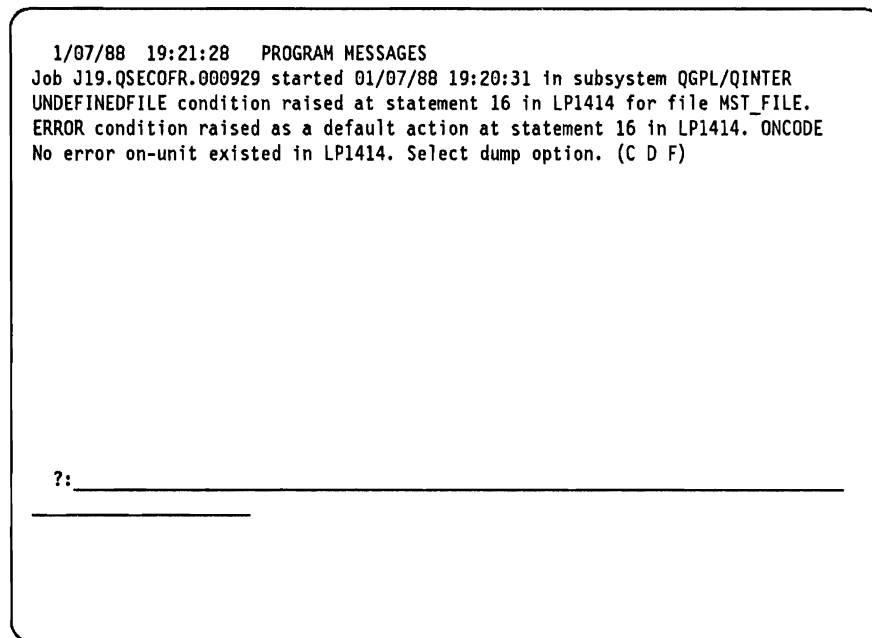


Figure 3-7. Error Dump Option Screen

To request a dump on the display, enter a D or F; the default is D. The output is on the system-wide dump file QPPGMDMP. The response D produces the same dump data as the PLIDUMP options T (trace), F (file information), and V (variables). The response F provides the same information as response D, and also produces a hexadecimal dump of the PL/I data spaces in which the program is running. The dump is the same as that produced by option H of PLIDUMP.

To cancel a dump, enter response C.

Using PLIIOFDB and PLIOPNFDB

You can obtain the contents of the system-defined input/output feedback area and the system-defined open feedback area by using the PLIIOFDB and PLIOPNFDB built-in subroutines. For a description of these subroutines, and a discussion of how to use them, see "PLIIOFDB Built-In Subroutine" on page 15-16 and "PLIOPNFDB Built-In Subroutine" on page 15-17.

Using ON Conditions

You can write your programs using specifiable ON conditions to monitor for problems you may encounter. ON conditions and condition codes are described in Appendix D, "Conditions and Condition Codes," and their use is discussed in Chapter 10, "Condition Handling Statements."

The figure below illustrates how you can use ON conditions in your program to alert you of problems.

USING ON CONDITIONS

```
5728PW1 R01M00 880715          SEU SOURCE LISTING          11/30/87 10:24:49          PAGE 1
SOURCE FILE . . . . . PLITST/PLISRC
MEMBER . . . . . ONCONDSEG
SEQNBR*...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 0
100  'ON ERROR' CONDITION
200  -----
300  DECLARE ONCODE BUILTIN;
400  .
500  .
600  .
700  ON ERROR
800  BEGIN;
900  ON ERROR SYSTEM;
1000 PUT FILE (SYSPRINT) SKIP(2) EDIT('** ERROR DETECTED **')(X(10),A);
1100 PUT FILE (SYSPRINT) SKIP EDIT('THE CONDITION CODE WAS ',ONCODE)
1200                                (X(13),A,F(4));
1300 END; /* BEGIN */
1400 .
1500 .
1600 .
1700 .
1800 -----
1900
2000 'ON ENDFILE' CONDITION
2100 -----
2200 DECLARE
2300 1 BIT_FLAGS STATIC,
2400 2 MORE_RECORDS BIT(1) ALIGNED,
2500 2 NO BIT(1) ALIGNED INIT('0'B),
2600 2 YES BIT(1) ALIGNED INIT('1'B);
2700 .
2800 .
2900 .
3000 ON ENDFILE (IN_FILE)
3100 MORE_RECORDS = NO;
3200 .
3300 .
3400 .
3500 MORE_RECORDS = YES;
3600 READ FILE (IN_FILE) INTO (INPUT_RECORD);
3700 .
3800 .
3900 .
4000
4100 -----
4200
4300 'ON ENDPAGE' CONDITION
4400 -----
4500 DECLARE PAGE_NUMBER BINARY FIXED(2);
4600 .
4700 .
4800 .
4900 PAGE_NUMBER = 1;
5000 .
5100 .
5200 .
5300 ON ENDPAGE (SYSPRINT)
```

Figure 3-8 (Part 1 of 3). Examples of ON conditions


```

5728PW1 R01M00 880715          SEU SOURCE LISTING          11/30/87 10:24:49          PAGE 2
SOURCE FILE . . . . . PLITST/PLISRC
MEMBER . . . . . ONCONDSEG
SEQNBR*...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 0
5400 BEGIN;
5500 PUT FILE (SYSPRINT) PAGE EDIT('PAGE ',PAGE_NUMBER)(X(81),A,F(2));
5600 PUT FILE (SYSPRINT) SKIP(2) EDIT('UPDATE REPORT')(X(30),A);
5700 PUT FILE (SYSPRINT) SKIP(2) EDIT('KEY ID','NAME','CUR BALANCE',
5800 'UPDATE AMOUNT','NEW BALANCE')(A,X(9),A,X(21),A,X(6),A,X(4),A);
5900 PAGE_NUMBER = PAGE_NUMBER + 1;
6000 END; /* BEGIN */
6100 .
6200 .
6300 .
6400 .
6500 -----
6600
6700 'ON KEY' CONDITION
6800 -----
6900 DECLARE ONCODE BUILTIN;
7000 .
7100 .
7200 .
7300 ON KEY (MST_FILE)
7400 BEGIN;
7500 ON ERROR SYSTEM;
7600 PUT FILE (SYSPRINT) SKIP(2) EDIT('** ERROR DETECTED **')(X(10),A);
7700 PUT FILE (SYSPRINT) SKIP EDIT('INVALID OPERATION INVOLVING KEY OF',
7800 ' MST_FILE. CONDITION CODE WAS ',ONCODE)(X(13),A,A,F(4));
7900 END; /* BEGIN */
8000 .
8100 .
8200 .
8300 .
8400 -----
8500
8600 'ON TRANSMIT' CONDITION
8700 -----
8800 DECLARE ONCODE BUILTIN;
8900 .
9000 .
9100 .
9200 ON TRANSMIT (MST_FILE)
9300 BEGIN;
9400 ON ERROR SYSTEM;
9500 PUT FILE (SYSPRINT) SKIP(2) EDIT('** UNEXPECTED ERROR ON I/O ',
9600 OPERATION OF MST_FILE')(X(10),A,A);
9700 PUT FILE (SYSPRINT) SKIP EDIT('THE FILE STATUS WAS ',ONCODE)
9800 (X(13),A,F(4));
9900 END; /* BEGIN */
10000 .
10100 .
10200 .
10300 .
10400 -----
10500
10600 'ON UNDEFINEDFILE' CONDITION (ALSO 'ON UNDF')

```

Figure 3-8 (Part 2 of 3). Examples of ON conditions

USING ON CONDITIONS

```
5728PW1 R01M00 880715          SEU SOURCE LISTING          11/30/87 10:24:49          PAGE 3
SOURCE FILE . . . . . PLITST/PLISRC
MEMBER . . . . . ONCONDSEG
SEQNBR*...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 0
10700 -----
10800 DECLARE ONCODE BUILTIN;
10900 .
11000 .
11100 .
11200 ON UNDEFINEDFILE (IN_FILE)
11300     BEGIN;
11400     ON ERROR SYSTEM;
11500     PUT FILE (SYSPRINT) SKIP(2) EDIT('** UNEXPECTED ERROR, UNDEFINED ',
11600     'FILE CONDITION RAISED ON OPENING OF IN_FILE')(X(10),A,A);
11700     PUT FILE (SYSPRINT) SKIP EDIT('THE CONDITION CODE WAS ',ONCODE)
11800     (X(13),A,F(4));
11900 END; /* BEGIN */
12000 .
12100 .
12200 .
12300 .
12400 -----
```

Figure 3-8 (Part 3 of 3). Examples of ON conditions

Part 2. Reference

The user's guide provides you with information on entering AS/400 PL/I programs, compiling and running these programs and finding errors in them. The reference information in Part 2 is provided if you need specific detailed information on PL/I compiler directives, references, expressions, statements, procedures, functions, subroutines, and pseudovariables.

The reference information is arranged as follows:

- Chapter 4, "Program Elements and Organization"
- Chapter 5, "PL/I Data Organization and Use"
- Chapter 6, "AS/400 PL/I File and Record Management"
- Chapter 7, "File Declaration and Input/Output"
- Chapter 8, "Using AS/400 Files"
- Chapter 9, "References and Expressions"
- Chapter 10, "Condition Handling Statements"
- Chapter 11, "Input and Output Statements"
- Chapter 12, "Declaring Names and Attributes of Variables"
- Chapter 13, "General PL/I Statements"
- Chapter 14, "Procedures, Subroutines, and Functions"
- Chapter 15, "Built-In Functions, Subroutines, and Pseudovariables."



Chapter 4. Program Elements and Organization

This chapter gives information on:

- PL/I statements and how to combine them into larger units:
 - Compound statements,
 - Do-groups
 - Blocks.
- The different types of blocks and how to combine them into a PL/I program.
- Names and how to declare and use them.

Characters That are Used in PL/I

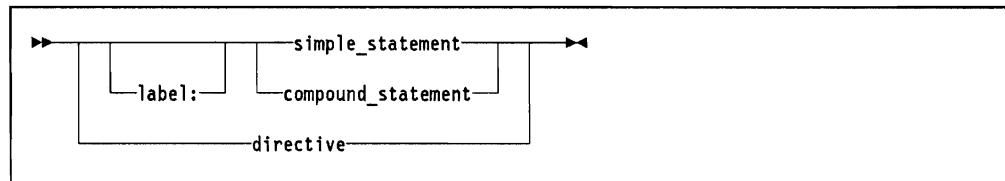
AS/400 PL/I uses the standard PL/I character set. A complete listing of the character set can be found in “EBCDIC Codes” on page B-15. Note that in AS/400 PL/I, you can use lowercase characters when creating a source program. The lowercase character is equivalent to its corresponding uppercase character except when used in comments and character literals.

PL/I Program Structure

A PL/I program is constructed from basic program elements called **statements** and **directives**. There can be up to 9999 statements in a program and up to 9999 sub-statements in a statement. There are two types of statements: simple and compound. Statements make up larger program elements called **do-groups** and **blocks**.

Statements and Directives

PL/I statements and directives are groupings of identifiers, constants, and delimiters. The syntax is:



Statement Labels

A **label** is an identifier that names a statement so that it can be referred to at some other point in the program. Statement labels are either label constants or entry constants (see “LABEL Attribute” on page 12-31 and “ENTRY Attribute” on page 12-32).

The descriptions of individual statements do not generally include the label. You can use a label for any statement unless it is explicitly stated that you cannot use one.

PL/I PROGRAM STRUCTURE

It is a good idea to use labels following the IF, THEN, ELSE, WHEN, and OTHERWISE keywords to make the program easy to read. The following example shows the format you should use:

```
        IF REMAINING_ITEMS = 0
            THEN
CASE1:    SIGNAL FINISH;
            ELSE
CASE2:    CALL NEXT_ITEM
```

Note: You cannot use a label with a compiler directive.

Simple Statements

There are three types of simple statements: **keyword**, **assignment**, and **null**. Each type ends with a semicolon.

A **keyword statement** begins with a keyword that indicates the function of the statement. For example:

```
READ FILE (INFILE) INTO (CURRENT_RECORD);
```

The **assignment statement** contains the assignment symbol (=). The statement does not begin with a keyword. For example:

```
A = B + C;
```

The **null statement** consists of a semicolon and may contain a label. For example:

```
LABEL:;
```

Directives

A directive consists of a % sign (or, optionally, an * for the %PROCESS directive) followed by an instruction to the compiler. Each directive ends with a semicolon. Labels are not allowed on directives. Directives are discussed in "Using Compiler Directives" on page 2-16.

Elements of a PL/I Statement

A PL/I statement consists of constants, identifiers, and delimiters.

Constants

A **constant** is a data item whose value cannot change. You refer to an arithmetic constant by directly representing the value of the constant, for example, 3.14.

Identifiers

An **identifier** consists of one or more alphabetic characters, and may contain digits, break characters `_` and the `$`, `#`, and `@` characters. An identifier must start with an alphabetic character and must not exceed 31 characters. The break character improves readability, as in `GROSS_PAY`.

If you use an identifier as a AS/400 program or file name, it must not exceed ten characters.

An identifier can be a user-defined name or a PL/I keyword, depending on how it is used.

User-Defined Names: A **user-defined name**, commonly called a **name**, is an identifier given to a variable or to a named constant. Any identifier can be used as a name.

At any point in a program, a name can have only one meaning. For example, you cannot use the same name for both a built-in function and a variable in the same block.

Examples of names are:

```
A
FILE2
LOOP_3
PAY_RATE
#32
```

Additional requirements for names are discussed later in this chapter.

PL/I Keywords: A **keyword** is an identifier that has a specific meaning when used in the predefined context. A keyword can specify an action to be taken or the attributes of data. Some examples are the keywords `READ`, `ENDFILE`, and `DECIMAL`. Some keywords can be abbreviated; the abbreviation is shown in the description of the individual keywords.

Delimiters

Delimiters are used to separate identifiers and constants. Delimiters, other than operators, are shown in Figure 4-1; operators are shown in Figure 4-2.

| Name | Delimiter | Use |
|-----------|-----------|-----------------------------------|
| operators | | See Figure 4-2 on page 4-4 |
| blank | | Separates elements of a statement |

Figure 4-1 (Part 1 of 2). Delimiters

PL/I PROGRAM STRUCTURE

| Name | Delimiter | Use |
|-------------------|--|---|
| comment | /*[text]*/ | Documents the program |
| comma | , | Separates elements of a list |
| period | . | Connects elements of a qualified name |
| semicolon | ; | Ends a statement |
| assignment symbol | = | Indicates assignment of a value. You can also use the character = as a comparison operator |
| colon | : | Connects a label prefix to a statement; delimits bounds in a dimension attribute |
| parentheses | () | Encloses a list, expression, or iteration factor; encloses information associated with various keywords |
| pointer | - > | Denotes a pointer qualifier |
| directive | %INCLUDE %PAGE %PROCESS *PROCESS %SKIP | Directs the compiler |

Figure 4-1 (Part 2 of 2). Delimiters

| Name | Operator | Use |
|------------|---------------|-----------------------------|
| Arithmetic | + | Addition or prefix plus |
| | - | Subtraction or prefix minus |
| | * | Multiplication |
| | / | Division |
| | ** | Exponentiation |
| Comparison | > | Greater than |
| | ↯ > | Not greater than |
| | > = | Greater than or equal to |
| | = | Equal to |
| | ↯ = | Not equal to |
| | < = | Less than or equal to |
| | < | Less than |
| ↯ < | Not less than | |
| Bit | ↯ | Not |
| | & | And |
| | | Or |

Figure 4-2 (Part 1 of 2). Operators

| Name | Operator | Use |
|--------|----------|---------------|
| String | | Concatenation |

Figure 4-2 (Part 2 of 2). Operators

The characters that can be used as delimiters can also be used in other contexts. For example, the period is a delimiter when used in structure qualification, such as A.B, but it is not considered a delimiter when used in a decimal constant, such as 3.14.

Blanks: You can surround each delimiter with blanks. One or more blanks must separate identifiers and constants that are not separated by another delimiter. In general, any number of blanks can appear wherever one blank is allowed.

Blanks cannot occur within identifiers, arithmetic and bit constants, or composite symbols. They are valid as data characters in character constants.

Other cases that require or permit blanks (for example, in GO TO or GOTO) are noted in the text where the feature of the language is discussed.

Some examples are:

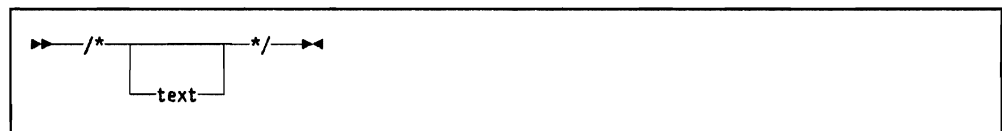
TABLE(10) is equivalent to TABLE (10)

FIRST,SECOND is equivalent to FIRST, SECOND

AB**BC is equivalent to AB ** BC

AB**BC is not equivalent to AB * * BC

Comments: You can use comments wherever blanks are allowed as delimiters in a program. A comment is treated as a blank and can therefore be used in place of a required separating blank. Comments do not affect the running of a program.



The composite symbol `/*` indicates the beginning of a comment and the composite symbol `*/` indicates its end. The text can contain any of the language or extralingual characters, except the `*/` composite symbol, which would end it.

An example of a comment in an assignment statement is:

`A = 1; /* INITIALIZE */`

The following example assigns a character constant to A; it does not contain a comment:

`A='/* THIS IS A CONSTANT,
NOT A COMMENT */';`

Program Organization

This section discusses how a PL/I program is organized and how control flows between blocks.

Programs

A PL/I program is a collection of one or more procedures, called **external procedures**, each of which can contain **internal procedures** or **begin-blocks** or both.

Activating a Program

A PL/I program becomes active when a calling program calls the **initial procedure**. This calling program may be a AS/400 Control Language (CL) program, or it could be a program written in another high level language. The initial procedure must be one of the external procedures of the program. In the following example:

```
CONTRL: PROCEDURE OPTIONS (MAIN);  
        DECLARE (PROC1,PROC2,PROC3) ENTRY EXTERNAL;  
        CALL PROC1;  
        CALL PROC2;  
        CALL PROC3;  
        END CONTRL;
```

the initial procedure is CONTRL; it calls external procedures PROC1, PROC2, and PROC3.

For more information about starting a PL/I program, see "Running the Program" on page 2-22.

Ending a Program

A program ends when the initial procedure ends. If a program ends normally or abnormally, control returns to the calling program.

Blocks

A block is the smallest delimited sequence of statements to which scoping and storage allocation rules apply.

There are two kinds of blocks: procedures and begin-blocks. The maximum number of blocks in any external procedure is equal to 255 minus the number of on-units in the procedure.

You can limit the scope of a name to a particular block (internal) or it can be known in all the blocks in a program (external). Storage may be allocated for a name only while a block is active (automatic) or while the program is running (static). You can define these attributes by explicit or implicit declarations within a block. (See "Names" on page 4-12 for more information about the scope of names and "STORAGE CONTROL" on page 5-15 for more information about the allocation of storage.)

You may find it easier to write and test a program by dividing it into blocks, particularly when a number of programmers are writing parts of the same program.

Some storage and some extra run time is used each time a block is activated. However, a program using multiple small blocks requires less storage to run, because storage for automatic variables is allocated on entry to the block, and is released on exit from the block.

Activating a Block

When an external procedure is called for the first time, storage is allocated for the static variables of all the blocks contained by the external procedure.

When an internal procedure or begin-block is activated:

- Array dimensions and string lengths of adjustable automatic variables which are not known at compile time are evaluated. The dimensions and lengths are those of the parameters passed to the procedure when it is called.
- Storage is allocated for automatic variables.

Begin-blocks and procedures are activated in different ways:

- Procedures other than the initial procedure are activated only when they are called by a procedure reference (see “Activating a Procedure” on page 4-10).
- Begin-blocks are activated through sequential flow (see “Activating a Begin-Block” on page 4-11) or by an on-unit.

Ending a Block

A procedure or begin-block can end in a number of ways, depending on the type of block. (See “Ending a Procedure” on page 4-11 and “Ending a Begin-Block” on page 4-12 for more information.)

When a block ends:

- The on-unit environment that existed before the block was activated is reestablished.
- Storage for all automatic variables allocated in the block is released.
- Static storage is released, and open files are closed if the block is the initial procedure of the program.

For more information on closing files, refer to “CLOSE Statement” on page 11-8.

Storage allocated for an automatic variable cannot be referred to after the block containing the declaration of the variable has ended. If such a reference is attempted (by means of a pointer variable to which the address of the automatic variable has been assigned, for example), the results are undefined. Similarly, the value of a label or internal entry constant cannot be referred to after the block containing its declaration has ended. If such a reference is attempted (by means of a label or entry vari-

PROGRAM ORGANIZATION

able to which the value has been assigned, for example), the results are undefined. Consider the following program:

```
MAINPROC: PROCEDURE OPTIONS (MAIN);
          DECLARE (ITEM1,ITEM2) FIXED DECIMAL (3);
          ITEM1 = 5;
          ITEM2 = 8;
BLOCK1: BEGIN;
          DECLARE BLOCKITEM
          FIXED DECIMAL (3,0) AUTOMATIC;
          BLOCKITEM = ITEM1;
          STMT1: ITEM1 = 0;
          END BLOCK1;
INVALID1: ITEM2 = BLOCKITEM;
INVALID2: GO TO STMT1;
          END MAINPROC;
```

When this program is compiled, the variable `BLOCKITEM` within the statement labelled `INVALID1` will be identified as being in error. The compiler will not recognize the variable name `BLOCKITEM`, because `BLOCKITEM` is declared in `BLOCK1`, and its scope does not include statement `INVALID1`. Similarly, the label `STMT1` in statement `INVALID2` will be identified as being in error.

If a `GO TO` statement transfers control out of a block, several blocks may be ended. If the label specified in the `GO TO` statement is contained in a block that did not directly activate the block being ended, all currently activated blocks in the activation sequence are ended. This is shown in the following example:

```
A: PROCEDURE;
  statement-a1
  statement-a2
  B: BEGIN;
    statement-b1
    statement-b2
    CALL C;
    statement-b3
    END B;
  statement-a3
  statement-a4
  C: PROCEDURE;
    statement-c1
    statement-c2
    statement-c3
    D: BEGIN;
      statement-d1
      statement-d2
      GO TO LAB;
      statement-d3
      END D;
    statement-c4
    END C;
  statement-a5
LAB: statement-a6
  statement-a7
  END A;
```

In this example, procedure A activates begin-block B, which activates procedure C, which activates begin-block D. In D, the statement GO TO LAB transfers control to statement-a6 in A. Because this statement is not contained in D, C, or B, all three blocks are ended; A remains active. Therefore, the transfer of control out of D ends intervening blocks B and C as well as block D.

Internal and External Procedures

A **procedure** is a sequence of statements that may be called for processing at one or more points in one or more programs within a run unit. The first statement is a PROCEDURE statement and the last is a corresponding END statement. (See "PROCEDURE Statement" on page 14-2 and "END Statement" on page 13-10.)

A procedure can be a subroutine or a function (see "Defining a Procedure" on page 14-1).

Any block can contain one or more blocks nested within it; that is, procedures and begin-blocks can contain other procedures and begin-blocks, which can contain others, and so on. A block must completely encompass any block contained within it.

A procedure can be external or internal. An **internal procedure** is contained in another block. An **external procedure** is not contained in another block.

Begin-blocks are always internal: they are always contained in another block. In the following example,

PROGRAM ORGANIZATION

```
A: PROCEDURE;  
.  
.  
.  
  B: BEGIN;  
    .  
    .  
    .  
    END B;  
  .  
  .  
  C: PROCEDURE;  
    .  
    .  
    D: BEGIN;  
      .  
      .  
      .  
      E: PROCEDURE;  
        .  
        .  
        .  
        END E;  
      .  
      END D;  
    END C;  
  .  
  .  
  END A;
```

procedure A is an external procedure because it is not contained in any other block. Block B is a begin-block that is contained in A; it contains no other blocks. Block C is an internal procedure; it contains begin-block D, which in turn contains internal procedure E. There are three levels of nesting relative to A: B and C are at a depth of one, D is at a depth of two, and E is at a depth of three.

The maximum depth of block nesting is 50.

Activating a Procedure

Normal sequential program processing ignores a procedure. Control passes directly from the statement immediately before the procedure's beginning to the statement immediately following the procedure's end.

A procedure is **activated** or **called** by an entry reference:

- Following the keyword **CALL** in a **CALL** statement (see "CALL Statement" on page 14-7).
- In a function reference (see "Function Reference" on page 14-4).

The point at which the entry reference appears is called the **point of calling**, and the block in which it appears is called the **calling block**. A calling block remains active when control is transferred to the called block.

When a procedure is called, processing begins with the first statement that can be processed. Processing is synchronous; that is, the calling procedure stops running until control is returned to it.

Communication between two procedures is by means of variables (“arguments”) passed from the calling procedure to the called procedure, by variables returned from the called procedure, and by names known within both procedures. Therefore, a procedure can operate upon different data when it is called at different times.

Ending a Procedure

A procedure ends when:

- A RETURN statement is processed within the procedure. Control then returns to the calling point in the calling procedure. If the calling point is a CALL statement, processing in the calling procedure resumes with the statement following the CALL. If the point of calling is a function reference, processing resumes with the statement containing the reference.
- The END statement of the procedure is reached. This is equivalent to a RETURN statement.
- A GO TO statement is processed and control is transferred out of the procedure. (The GO TO statement is discussed under “GO TO Statement” on page 13-11.)
- A STOP statement is processed. This also ends the run unit.
- A condition is raised and the implicit action ends the procedure. This also ends the run unit.

Begin-Blocks

A **begin-block** is a sequence of statements delimited by a BEGIN statement and a corresponding END statement.

A label is optional for a begin-block.

Activating a Begin-Block

Begin-blocks are activated through normal flow or by error-handling on-conditions. In general, they can appear wherever a single statement can appear.

When a begin-block is activated, the encompassing block or blocks remain active.

Control can be transferred to a labeled BEGIN statement by means of a GO TO statement.

Ending a Begin-Block

A begin-block ends when:

- Control reaches the END statement for the block. Control is then transferred to the statement following the END statement. (See “Running an On-Unit” on page 10-3 for a discussion of normal return from an on-unit.)
- A STOP statement is processed. This also ends the run-unit.
- A condition is raised and the implicit action ends the run-unit.
- A GO TO statement is processed and control is transferred to a point outside of the block.
- A RETURN statement is processed and control is transferred out of both the begin-block and its containing procedure.

Names

You refer to each variable, and each file, label, and entry constant in a PL/I program by a name.

Each name and its attributes must be made known in the block in which it is used by either an explicit or a contextual declaration.

A name need not have the same meaning throughout a program. A name declared within a block has a meaning only within that block. Outside the block, it is unknown unless the same name is also declared in the outer block. The name in the outer block refers to a different data item. You can specify local definitions and write a block (a procedure or a begin-block) without knowing all the names being used in other blocks.

The part of the program to which a name applies is called the **scope** of that name. Each declaration of a name establishes a scope for it.

To understand the rules for the scope of a name, you need to know the meaning of the terms “contained in” and “internal to.”

Everything in a block, from the PROCEDURE or BEGIN statement through to the corresponding END statement, is **contained in** that block. However, the label of the BEGIN or PROCEDURE statement that heads the block is not contained in that block. Nested blocks are contained in the block in which they appear.

Elements contained in a block, but not contained in any block nested within it, are **internal to** that block. Consider the following example:

```
PROC1: PROCEDURE;
    STMT1: INTEGER1 = SQRT(INTEGER2);
    PROC2: PROCEDURE;
        STMT2: INTEGER3 = INTEGER1;
    END PROC2;
END PROC1;
```


STMT1 and STMT2 are both contained in PROC1. STMT2 is also contained in PROC2. STMT1 is internal to PROC1. STMT2 is internal to PROC2.

The entry name of an internal procedure or the label of a BEGIN statement is internal to the containing block. The entry name of an external procedure is not internal to the external procedure.

Explicit Declaration of a Name

A name is explicitly declared if it appears:

- In a DECLARE statement. The DECLARE statement explicitly declares attributes of names.
- In a parameter list. The appearance of the name in a parameter list constitutes an explicit declaration of the name as a parameter of the containing procedure. The attributes for this parameter must be in a DECLARE statement internal to the same procedure.
- As the label prefix of a PROCEDURE statement. A labeled PROCEDURE statement constitutes a declaration, within the containing block, of the procedure name as an entry constant.
- As the label prefix of a statement other than a PROCEDURE statement. The label prefix constitutes an explicit declaration of a label constant within the containing block.

Note: An explicit declaration overrides a contextual declaration.

The scope of an explicit declaration of a name is the block it is internal to. This includes all contained blocks. This does not include blocks that have another explicit declaration of the same name internal to them.

The syntax and use of the DECLARE statement is described in Chapter 12, "Declaring Names and Attributes of Variables."

Contextual Declaration of a Name

Only built-in function and built-in subroutine names can be declared contextually. To contextually declare a name as a built-in function name, it must appear as a reference and be followed by a parenthesized argument list. To contextually declare a name as a built-in subroutine name, it must appear as a reference in a subroutine call.

Contextual declaration of a built-in function or built-in subroutine name has the same effect as if the name was declared in the external procedure, even when the statement that causes the contextual declaration is internal to another block that is contained in the external procedure. Consequently, the scope of the contextual declaration is the entire external procedure, except for any blocks in which the name is explicitly declared.

NAMES

Multiple Declarations of Names

Multiple declarations are not valid. They occur when two or more declarations of the same name are internal to the same block. Multiple declarations are valid when at least one of the names is declared within a structure in such a way that structure qualification can be used to make references unique.

Scopes of Names

Figure 4-3 is a sample procedure that illustrates the scopes of data declarations. The brackets to the left indicate the block structure; the brackets to the right show the scope of each declaration of a name. The scopes of the two declarations of Q are shown as Q and Q'.

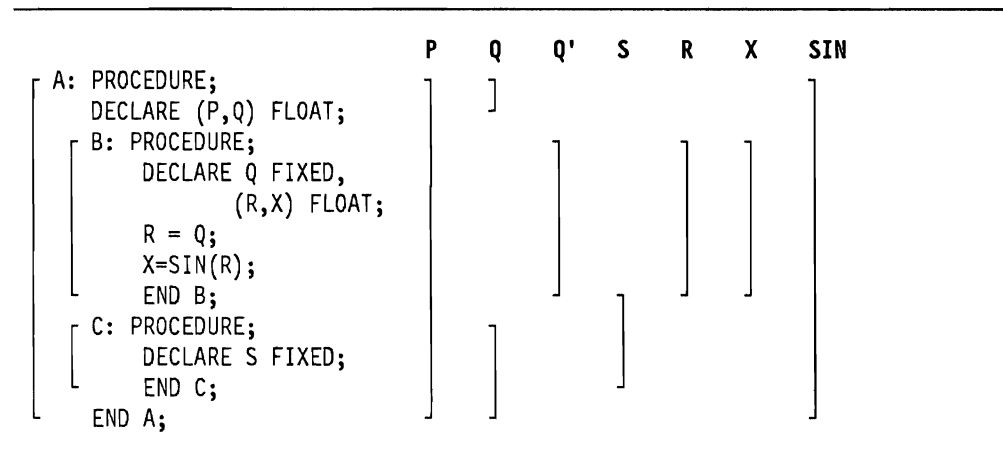


Figure 4-3. Scopes of Data Declarations

P is declared in block A and known throughout A.

Q is declared in block A and in block B. The scope of the first declaration of Q is all of A except B; the scope of the second declaration of Q (Q') is block B only.

SIN is referred to in block B. This results in a contextual declaration in the external procedure A. This declaration therefore applies to all of procedure A, including its contained procedures B and C.

S is explicitly declared in procedure C and is known only within C. R and X are declared in block B and are known only within block B.

Figure 4-4 on page 4-15 illustrates the scopes of entry constant and statement label declarations. The example shows two external procedures.

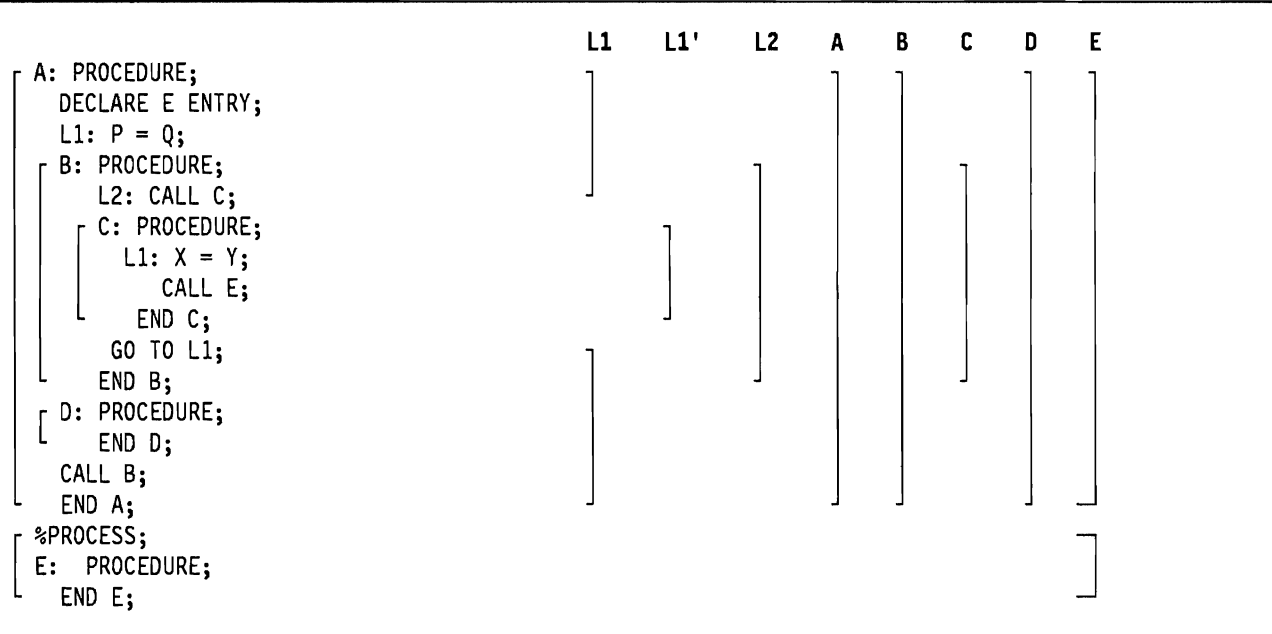


Figure 4-4. Scopes of Entry and Label Declarations

E is explicitly declared in A as an external entry constant. The explicit declaration of E in block A applies throughout block A; its explicit declaration as the entry constant of block E applies throughout block E. The scope of the name E is all of block A and all of block E. The scope of the name A is all of the block A only, and not block E.

The label L1 appears on statements internal to A and to C. Two separate declarations are therefore established; the first applies to all of block A except block C, and the second applies to block C only. Therefore, when the GO TO statement in block B is processed, control is transferred to L1 in block A, and block B is ended.

B and D are explicitly declared in block A and can be referred to from anywhere within A. Because they are internal, however, they cannot be referred to in block E.

C is explicitly declared in B and can be referred to from within B, but not from outside B.

L2 is declared in B and can be referred to in block B, including C, which is contained in B, but not from outside B.

If a PL/I keyword and its abbreviation are both declared as user-defined names in a program, the scopes of the two declarations may be different. For example:

NAMES

```
A: PROCEDURE;  
  DECLARE DEC FIXED DECIMAL (6),  
         DECIMAL FILE;  
B: BEGIN;  
  DECLARE DEC BUILTIN,  
         (Y,Z) FIXED DECIMAL (6);  
  Y=DEC(Z,6);  
  END B;  
CLOSE FILE (DECIMAL);  
END A;
```

DEC is known as a fixed-point decimal data item in block A, and as a built-in function in block B, where it is declared again. DECIMAL is known as a file throughout blocks A and B.

Using the Scope Attributes

You can use the INTERNAL and EXTERNAL attributes to specify the scope of a name.

For a description of the syntax of the scope attributes, INTERNAL and EXTERNAL, see "Scope Attributes" on page 12-40. For a description of how PL/I handles INTERNAL and EXTERNAL files, see "Scoping of Open Files (File Sharing)" on page 7-11.

INTERNAL specifies that the name is known only in the declaring block. The scope of the name is the same as the scope of its declaration. Any other explicit declaration of that name refers to a new object with a different, non-overlapping scope.

A name with the EXTERNAL attribute can be declared in more than one external procedure. It is linked across external procedures. No external name can be declared more than once in the same external procedure. The scope of the name includes the scopes of all the declarations of that name (with the EXTERNAL attribute) within the run unit.

External names cannot exceed ten characters in length.

Different declarations of the same name with the EXTERNAL attribute must have identical attributes after any defaults have been applied.

When you declare a major structure name as EXTERNAL in more than one external procedure, the attributes of the structure members must be the same, although the corresponding member names need not be identical. For example:

```

PROCA: PROCEDURE;
      DECLARE 1 A EXTERNAL,
              2 B FIXED,
              2 C FLOAT;
      .
      .
      .
END PROCA;

PROCB: PROCEDURE;
      DECLARE 1 A EXTERNAL,
              2 B FIXED,
              2 D FLOAT;
      .
      .
      .
END PROCB;

```

In this example, if A.B is changed in PROCA, it is also changed in PROCB, and if it is changed in PROCB, it is also changed in PROCA. If A.C is changed in PROCA, A.D is changed in PROCB, and if A.D is changed in PROCB, A.C is changed in PROCA.

The attribute listing, which is available as optional output from the compiler, helps to check the use and attributes of names. The following program example illustrates the use and attributes of names, and the rules for scopes of names:

```

A: PROCEDURE;
  DECLARE S CHARACTER (21),
          M FIXED DECIMAL (7),
          N BINARY (15);
  DECLARE SYSIN FILE INPUT;
  DECLARE SET ENTRY(FIXED DECIMAL(1));
  CALL SET (3);
E: GET FILE(SYSIN) EDIT (S,M,N)
    (A(21),F(7),F(3));
B: BEGIN;
  DECLARE X(M,N) FIXED DECIMAL (7),
          Y(M) FIXED DECIMAL (7);
  GET FILE(SYSIN) EDIT (X,Y)
    (F(7),F(7));
  CALL C(X,Y);
C: PROCEDURE (P,Q);
  DECLARE (I,J) FIXED BINARY (15)
          INTERNAL,
          P(*,*) FIXED DECIMAL (7),
          Q(*) FIXED DECIMAL (7),
          FIXED BINARY (15)
          EXTERNAL,
          OUT ENTRY(LABEL),
          SUM FIXED DECIMAL (9);
  S = 0;
  DO I = 1 TO M;
    SUM = 0;
    DO J = 1 TO N;

```

NAMES

```
        SUM = SUM + P(I,J);
    END;
    IF SUM = Q(I) THEN
        GO TO B;
    S = S + 1;
    IF S = 3 THEN
        CALL OUT (E);
    CALL D(I);
B: END;
    END C;
D: PROCEDURE (N);
    DECLARE N          FIXED BINARY (15);
    DECLARE SYSPRINT   FILE OUTPUT;
    PUT FILE(SYSPRINT) EDIT
        ('ERROR IN ROW' ,
        N, 'TABLE NAME ', S)
        (A(12),F(4),X(1),
        A(11),A(21));
    END D;
    END B;
    GO TO E;
    END A;
%PROCESS;
    OUT: PROCEDURE (R);
        DECLARE R      LABEL,
            (M,L) FIXED DECIMAL (7)
            STATIC INTERNAL
            INITIAL (0),
            S          FIXED BINARY (15)
            EXTERNAL;

        M = M+1;
        S = 0;
        IF M<L THEN STOP;
        ELSE GO TO R;
    END OUT;
%PROCESS;
    SET: PROCEDURE(Z);
        DECLARE Z      FIXED DECIMAL(1),
            L          FIXED DECIMAL(1) STATIC

        L=Z;
        RETURN;
    END SET;
```

A is an external name. The scope of A is all of block A, plus any other blocks where A is declared as external.

S is declared in block A and block C, as well as in block OUT. The declarations of S in block C and in block OUT declare S as external. They specify identical attributes for S, and declare the same external variable. The declaration of S in block A introduces a different, internal variable.

Within external procedure A, the character variable declaration of S applies to all of block A except block C. The fixed binary declaration of S applies only within block

C. Although D is called from within block C, the reference to S in the PUT statement in D is to the character variable S and not to the S declared in block C.

N is a parameter in block D, but is also declared in block A. These two declarations on the name N refer to different objects, although, in this case, the objects have the same attributes, which are `BINARY FIXED(15)` and `INTERNAL`.

X and Y are known throughout B and could be referred to in block C or D within B, but not in that part of A outside B.

P and Q are parameters and therefore require explicit declaration. Although the arguments X and Y are declared as arrays and are known in block C, it is still necessary to declare P and Q in a `DECLARE` statement to establish that they, too, are arrays. (Asterisks indicate that the bounds of the parameters are taken from the corresponding arguments.)

I and J are known only in block C. M is known throughout block A, including all its contained blocks.

The second external procedure in the example has the entry name `OUT`, and the third external procedure has the name `SET`. The entry constants `SET` and `OUT` get the attributes `ENTRY` and `EXTERNAL` and are known throughout their external procedures. Because these external procedures are referenced in the external procedure A, they must be declared with an appropriate `ENTRY` attribute in procedure A.

The label prefix B appears twice in the program. It is first declared explicitly by its appearance as the label of a `begin-block A`. It is declared again as a label within block C by its appearance as a prefix to an `END` statement. The `GO TO B` statement within block C, therefore, refers to the label on the `END` statement within block C. Outside block C, any reference to B would be to the label of the `begin-block`.

C and D can be called from any point within B, but not from that part of A outside B or from another external procedure like `OUT` or `SET`. Similarly, because E is known throughout the external procedure A, a transfer can be made to E from any point within A. The label B within block C, however can only be referred to from within C.

Control can be transferred out of a block and back to a block that was activated, by means of a `GO TO` statement. In the external procedure `OUT`, the label E from block A is passed as an argument to the label parameter R. The statement `GO TO R` causes control to pass to the label E, although E is declared within A and is not within `OUT`.

The variables M and L are declared as `STATIC` within the procedure `OUT`; their values are preserved between calls to `OUT`.

NAMES



Chapter 5. PL/I Data Organization and Use

This chapter contains information on how data is stored and manipulated. Data items come in two forms: scalars, or aggregates. You can use alignment and mapping attributes as well as the DECLARE statement to control how this data is stored. There are three data types: arithmetic, character, and bit. You can assign data items to one of these types and convert between types using built-in conversion functions.

DATA ORGANIZATION

Data items can be single data items, called **scalars**, or they can be grouped together to form **data aggregates**, in which they can be referred to either individually or collectively. Data aggregates can be **arrays** or **structures**. A variable that represents a single data item is a **scalar variable**. A variable that represents an aggregate of data items is either an **array variable** or a **structure variable**.

Any type of problem data variable or program control variable can be grouped into arrays or structures.

Using Arrays and the Dimension Attribute

An **array** is a collection, into one or more dimensions, of one or more array-elements with identical attributes. An **array-element** can be a scalar variable or a structure. Only the array itself is given a name. An individual item of an array is referred to by the array name and a subscript giving the item's position inside the array.

An array is declared with the dimension attribute, which defines the subscript format. For a description of the syntax of the dimension attribute, see "Arrays and the Dimension Attribute" on page 12-38.

Examples of Array Declarations

```
DECLARE LIST(8) FIXED DECIMAL (3);
```

In the example above, LIST is declared a one-dimensional array of eight elements, each of which is a fixed-point decimal element of three digits. The single dimension of LIST has bounds of 1 and 8; its extent is 8.

```
DECLARE TABLE1(4,2) FIXED DECIMAL (3);
```

TABLE1 is declared a two-dimensional array, also of eight fixed-point decimal elements. The two dimensions of TABLE1 have bounds of 1 and 4, and 1 and 2; the extents are 4 and 2.

```
DECLARE TABLE2(10,1:8) FIXED DECIMAL (6,2);
```

TABLE2 is declared a two-dimensional array of eighty fixed-point decimal elements, each with six digits, of which two are to the right of the decimal point. The two

DATA ORGANIZATION

dimensions of TABLE2 have bounds of 1 and 10, and 1 and 8; the extents are 10 and 8.

In AS/400 PL/I, the only lower bound you can specify is 1.

```
DECLARE INDEX1 FIXED BINARY (15) STATIC INITIAL (8);  
DECLARE LIST(1:INDEX1) FIXED DECIMAL (4);
```

The bounds of LIST are 1 and INDEX1, with INDEX1 initialized as 8.

The following example shows a factored array declaration:

```
DECLARE (A,B,C,D)(10) BINARY FIXED;
```

The variables A, B, C, and D are to represent one-dimensional arrays, each consisting of ten fixed-point binary items with a default length of 15.

Subscripts

The dimensions of an array determine the way the elements of the array are referred to. For example, the array LIST, which is declared above as a one-dimensional array, can be considered as a linear sequence of eight elements. If the contents of the elements of the array are

20 5 10 30 630 150 310 70

in that order, they can be referred to as follows:

| Reference | Element |
|-----------|---------|
| LIST(1) | 20 |
| LIST(2) | 5 |
| LIST(3) | 10 |
| LIST(4) | 30 |
| LIST(5) | 630 |
| LIST(6) | 150 |
| LIST(7) | 310 |
| LIST(8) | 70 |

Each of the numbers following LIST is a **subscript**. A subscript identifies a particular element of the array. A reference to a subscripted name, such as LIST(4), refers to a single item. In the example, LIST(4) has a value of 30. The entire array can be referred to by the name of the array, with no following subscript. For example, all of the elements of LIST could be set to zero by the statement LIST = 0.

The same data could be organized in the two-dimensional array TABLE1 declared above. TABLE1 could then be considered as a matrix of four rows (m) and two columns (n), as follows:

| n m | 1 | 2 |
|--------|-----|-----|
| 1 | 20 | 5 |
| 2 | 10 | 30 |
| 3 | 630 | 150 |
| 4 | 310 | 70 |

TABLE1 is referred to by a subscripted name with two parenthesized subscripts, separated by a comma. For example, TABLE1 (2,1) specifies the first item in the second row, which is 10.

The use here of a matrix to illustrate TABLE1 bears no relationship to the way in which the items are actually organized in storage. Elements of an array are stored in row major order with the right-most subscript varying most rapidly. For example, the array TABLE1 is stored in the order TABLE1(1,1), TABLE1(1,2), TABLE1(2,1), TABLE1(2,2) and so on.

A subscripted reference to an array must contain as many subscripts as there are dimensions in the array. (See "Arrays of Structures" on page 5-5 for arrays with inherited dimensions.)

The examples in this chapter have arrays of arithmetic data. Variables of any data type except FILE can be collected into an array. String arrays (character or bit) are valid, as are arrays of label, entry, or pointer data.

Expressions as Subscripts

Any integer expression can be used as a subscript. The expression is converted to fixed-point binary with a precision of 31. Therefore, TABLE(I,J*K) could refer to the different elements of TABLE by varying the values of the integers I, J, and K.

Using Structures and Level Numbers

A **structure** is a collection of data items that need not have identical attributes.

Like an array, the entire structure is given a name, which can be used to refer to the entire aggregate of data. But, unlike an array, each field of a structure also has a name.

A structure has different **levels**. At the first level is the **major structure**; at lower levels are the **minor structures**; and at the lowest are the **fields** of the structure. A structure field can be a scalar variable or an array.

The members at the next lower level of a structure or substructure are the **immediate components** of the structure or substructure.

DATA ORGANIZATION

You specify the organization of a structure in a DECLARE statement by placing level numbers before the names, as described in "Structures and Level Numbers" on page 12-39. The major structure name must be declared with the level number 1, and minor structures and field names with level numbers greater than 1; level numbers must be integer constants.

For example, the items of a payroll could be declared as follows:

```
DECLARE 1 PAYROLL,  
        2 EMPLOYEE_NO  FIXED DECIMAL (7),  
        2 NAME,  
          3 LAST      CHARACTER (15),  
          3 FIRST     CHARACTER (15),  
        2 HOURS,  
          3 REGULAR   FIXED DECIMAL (5,2),  
          3 OVERTIME  FIXED DECIMAL (5,2),  
        2 RATE,  
          3 REGULAR   FIXED DECIMAL (5,2),  
          3 OVERTIME  FIXED DECIMAL (5,2);
```

In this example, PAYROLL is a major structure with the immediate components EMPLOYEE_NO, NAME, HOURS, and RATE. EMPLOYEE_NO is a field; NAME, HOURS, and RATE are minor structures, each containing two fields. You can refer to the entire structure by the name PAYROLL, to portions of the structure by the minor structure names, or to a field by the name of the field.

The level numbers you choose for successively deeper levels need not be the immediately succeeding integers. A minor structure at level n contains all the names with level numbers greater than n that lie between that minor structure name and the next name with a level number less than or equal to n .

PAYROLL could have been declared as follows:

```
DECLARE 1 PAYROLL,  
        4 EMPLOYEE_NO  FIXED DECIMAL (7),  
        3 NAME,  
          5 LAST      CHARACTER (15),  
          5 FIRST     CHARACTER (15),  
        2 HOURS,  
          6 REGULAR   FIXED DECIMAL (5,2),  
          5 OVERTIME  FIXED DECIMAL (5,2),  
        2 RATE,  
          45 REGULAR  FIXED DECIMAL (5,2),  
          30 OVERTIME FIXED DECIMAL (5,2);
```

This declaration would result in exactly the same structuring as that of the previous declaration.

Therefore, there is a difference between logical level and level number. The item with the greatest level number is not necessarily the item with the deepest logical level. But if the structure declaration is written with consistent level numbers and suitable indentation, the logical levels are immediately apparent.

You can, in any case, determine the logical level of each item in the structure by applying the following rule to each item in turn, starting at the beginning of the structure declaration: the logical level of a given item is always one unit deeper than that of its immediate containing structure. For example, in the first declaration of PAYROLL, the logical levels and level numbers are the same. In the second declaration, the level numbers are different, but the logical levels are the same as in the first declaration.

The description of a major structure is ended by the declaration of another item with the level number 1, by the declaration of another item with no level number, or by the end of the DECLARE statement or descriptor list.

The maximum depth of logical levels is 15, and the highest valid level number is 255. The maximum length of a structure is 32 767 bytes.

Structure-Qualification

A minor structure can be referred to by the minor structure name alone and a structure field by the field name alone if there is no ambiguity. In the PAYROLL example, a reference to either REGULAR or OVERTIME would be ambiguous without structure-qualification to make the reference unique.

A **qualified reference** is a field name or a minor structure name that is qualified with one or more names at a higher level, connected by periods. Blanks may appear on either side of the period.

Structure-qualification is in the order of levels; that is, the name at the highest level must appear first, with the name at the deepest level appearing last.

Names within a structure need not be unique within the block in which they are declared. Also, one or more qualifying names can be omitted, provided that the name or names used identify a single field or minor structure. The qualified references PAYROLL.LAST and NAME.LAST, for example, are both equivalent to the name PAYROLL.NAME.LAST.

Arrays of Structures

A structure name, either major or minor, can be given a dimension attribute in a DECLARE statement to declare an array of structures. An **array of structures** is an array whose elements are structures that have identical names, levels, and element attributes.

For example, if you were to use a structure, WEATHER, to process meteorological information for each month of a year, you might declare it as:

DATA ORGANIZATION

```
DECLARE 1 WEATHER(12),
        5 TEMPERATURE,
          10 HIGH      DECIMAL FIXED (5,1),
          10 LOW       DECIMAL FIXED (3,1),
        5 WIND_VELOCITY,
          10 HIGH      DECIMAL FIXED (3),
          10 LOW       DECIMAL FIXED (3),
        5 PRECIPITATION,
          10 TOTAL     DECIMAL FIXED (3,1),
          10 AVERAGE  DECIMAL FIXED (3,1);
```

You could then refer to all the weather data for July by specifying WEATHER(7) and to the particular aspects of the July weather by TEMPERATURE(7) and WIND_VELOCITY(7). The specifications PRECIPITATION.TOTAL(7) and TOTAL(7) would both refer to the total precipitation during the month of July.

TEMPERATURE.HIGH(3), which would refer to the high temperature in March, is a **subscripted qualified reference**.

The need for subscripted qualified references becomes more apparent when an array of structures contains an array of minor structures. For example, consider the following array of structures:

```
DECLARE 1 A (6,6),
        5 B (5),
          10 C FIXED,
          10 D FIXED,
        E FIXED;
```

Both A and B are arrays of structures. To reference a data item, it may be necessary to use as many as three names and three subscripts.

You must include the subscripts to the right of the name or qualified list of names. For example, A.B.C(1,1,2) is valid, whereas A(1,1).B(2).C is not. A.B.C(1,1,2) references a particular C that is in an element of B in a structure in A.

Any item declared within an array of structures **inherits** dimensions declared in the containing structure. For example, in the above declaration for the array of structures A, B is a three-dimensional array of structures, because it inherits the two dimensions declared for A. If B is unique and requires no qualification, any reference to a particular element of B would require three subscripts: two to identify the specific structure in A and one to identify the specific element of B within that structure in A.

A reference to an array with inherited dimensions must be subscripted, and the number of subscripts must equal the number of inherited dimensions or the total number of dimensions of the array. Therefore, with the declaration above, A, A(1,2), B(1,2), B(1,2,3) and C(1,2,3) are valid references, but B, C, and C(1,2) are not.

Performance Considerations with Large Aggregates

Programs which process large `STATIC` or `AUTOMATIC` aggregates (those approaching or exceeding 32 767 bytes in size) may take a long time to run. The reason for this is that the machine brings into main memory all static and automatic work areas before running the program. If the work areas become too large, the excessive paging results in system overhead which can significantly increase the time required to run the program.

You can reduce this paging activity by declaring large aggregates as `BASED` variables and using the `ALLOCATE` statement (see “`ALLOCATE` Statement for Based Variables” on page 5-22) before referencing the variables. Allocated storage for `BASED` variables is paged into main memory on a demand “paging” basis: only the pages that the program references are paged in.

Data Alignment and the Alignment Attributes

Data is stored in units of eight bits, or a multiple of eight bits. Each eight-bit unit of information is called a **byte**.

Bytes may be grouped together in units of information as a halfword (two bytes), a word (four bytes; also called a fullword), a doubleword (eight bytes), or a quadword (16 bytes), starting at an integral boundary for that unit. An **integral boundary** for a unit is at a multiple of that unit: a byte boundary can be at any byte, a halfword boundary at a multiple of two bytes, a word boundary at a multiple of four bytes, a doubleword at a multiple of eight bytes, and a quadword at a multiple of 16 bytes.

Byte locations in storage are consecutively numbered, starting with zero. Each number is considered the address of the corresponding byte. A group of bytes in storage is addressed by the leftmost byte of the group.

A field aligned on a halfword or word boundary can be accessed faster than a field of the same length that is not aligned on an integral boundary. For some operations, the data used must be aligned on its integral boundary.

Data items can be aligned on their integral boundaries to give the fastest possible processing. But this is not always desirable, as there may be unused bytes between successive data items, which increases the use of storage. This is particularly important when the data items are members of aggregates that are used to create a file, because the unused bytes will increase the amount of external storage required. Consequently, although the `UNALIGNED` attribute may increase run time, it can reduce storage requirements.

By means of the `ALIGNED` and `UNALIGNED` attributes, you can choose to align data on the appropriate integral boundary.

`ALIGNED` specifies that the data item is aligned on the storage boundary corresponding to its data type requirement. For example, `BIN (15)` data is aligned on a halfword boundary and `BIN (31)` data on a fullword boundary. See “Data Mapping” on page 5-9 for a definition of these requirements.

DATA ALIGNMENT

UNALIGNED specifies that the data need not be aligned. The compiler may generate code that moves the data to an appropriate integral boundary before an operation is processed, if the operation requires data alignment.

Bit data must be declared as ALIGNED. Aligned bit strings start at a byte boundary. Consider the following example:

```
DECLARE BITSTRING1  BIT (3) ALIGNED,  
        BITSTRING2  BIT (5) ALIGNED;
```

BITSTRING1 starts on a byte boundary. BITSTRING2 starts on the next byte boundary. The five bits intervening between BITSTRING1 and BITSTRING2 are not addressable by the program.

The default for character data and picture data is UNALIGNED. For all other data types, the default is ALIGNED. If you specify UNALIGNED with character data, the compiler will issue an error message. If you specify ALIGNED with non-varying character data, the specification of ALIGNED will be ignored, and the compiler will issue an error message.

You can specify alignment attributes for scalars and arrays only.

ALIGNED can be specified for character varying data and must be specified for bit data.

UNALIGNED can be specified for fixed-point binary, floating-point binary, and floating-point decimal data.

The following example illustrates explicit and default alignment.


```

DECLARE 1 SAMPLESTRUCTURE,
      5 BIT1  BIT (3) ALIGNED,
        /* ALIGNED EXPLICITLY */

      5 MINORSTRUCTURE1,
      10 BIT2  BIT (5) ALIGNED,
        /* ALIGNED EXPLICITLY */

      10 MINORSTRUCTURE2,
      15 BINFLT1  FLOAT UNALIGNED,
        /* UNALIGNED EXPLICITLY */

      15 DECFIXED  FIXED DECIMAL,
        /* ALIGNED BY DEFAULT */

      15 CHAR1  CHARACTER (1),
        /* UNALIGNED BY DEFAULT */

      15 CHAR2  CHARACTER(2)
                VARYING ALIGNED,
        /* ALIGNED EXPLICITLY */

      10 BINFIXED1  FIXED,
        /* ALIGNED BY DEFAULT */

      5 PIC1  PICTURE '99.V9';
        /* UNALIGNED BY DEFAULT */

```

Data Mapping

This section describes the mapping of data onto storage. In general, you need not know the precise rules for data mapping; the compiler can print an aggregate length table of all the arrays and major structures in the source program. (You can print this table using the `*AGGREGATE` option of the `OPTION` parameter for the `CL` command `CRTPLIPGM`. See Chapter 2, "Creating, Compiling, and Running Your PL/I Program" for more information.) However, you may want to know the rules for data mapping in the following cases:

- To determine record lengths and alignments when you use record data transmission.
- To determine the correspondence of pointers and based variables.
- When using the `UNSPEC` built-in function or pseudovisible.

A unit of data is mapped onto storage by determining the alignment (`a`), displacement (`d`), and length (`l`) of the unit, where:

`a` is the alignment boundary:

- `a = 1` for byte alignment
- `a = 2` for halfword alignment
- `a = 4` for word alignment
- `a = 8` for doubleword alignment.

DATA MAPPING

a = 16 for quadword alignment.

d is the displacement, in bytes, of the start of the unit from the alignment boundary.

l is the length, in bytes, of the unit.

Scalar Data Mapping

The scalar data mapping algorithm derives immediately from Figure 5-1. The values of "a" and "l" are determined from the appropriate variable type in the figure. The value of "d" is always 0.

Examples:

```

DECLARE A BINARY FIXED (15) ALIGNED
  where: a=2, d=0, l=2
DECLARE B BINARY FIXED (12) ALIGNED
  where: a=2, d=0, l=2
DECLARE C BINARY FIXED (31) UNALIGNED
  where: a=1, d=0, l=4
DECLARE A BIT (2) ALIGNED
  where: a=1, d=0, l=1
    
```

| Variable Type | Stored internally as | Length (in bytes) | ALIGNED boundary | UNALIGNED boundary |
|-----------------------|--|---|------------------|--------------------|
| BIT (n) | One byte for each group of 8 bits (or part thereof) | $\text{ceil}(n/8)$ | Byte | Not applicable |
| CHARACTER (n) | One byte per character | n | Not applicable | Byte |
| PICTURE | One byte for each PICTURE character (except V) | Number of PICTURE characters (other than V) | Not applicable | Byte |
| DECIMAL FIXED (p,q) | Packed decimal format (1/2 byte per digit, plus 1/2 byte for sign) | $\text{ceil}((p+1)/2)$ | Byte | Not applicable |
| CHARACTER (n) VARYING | One byte per character | n+2 | Halfword | Byte ¹ |

Figure 5-1 (Part 1 of 2). Storage and Alignment Requirements of Scalar Data

| Variable Type | Stored internally as | Length (in bytes) | ALIGNED boundary | UNALIGNED boundary |
|---|-------------------------|-------------------|---|--------------------|
| BINARY FIXED (p) $1 \leq p \leq 15$ | Halfword binary integer | 2 | Halfword ¹ | Byte |
| BINARY FIXED (p) $16 \leq p \leq 31$ | Fullword binary integer | 4 | Word ¹ | Byte |
| BINARY FLOAT (p) $1 \leq p \leq 24$ | Short floating-point | 4 | Word ¹ | Byte |
| DECIMAL FLOAT (p) $1 \leq p \leq 7$ | Short floating-point | 4 | Word ¹ | Byte |
| POINTER | — | 16 | Quadword | Not applicable |
| ENTRY | — | 16 | Halfword | Not applicable |
| LABEL | — | 16 | Halfword | Not applicable |
| BINARY FLOAT (p) $25 \leq p \leq 53$ | Long floating-point | 8 | Doubleword within structures; otherwise word ¹ | Byte |
| DECIMAL FLOAT (p) $8 \leq p \leq 16$ | Long floating-point | 8 | Doubleword within structures; otherwise word ¹ | Byte |

Figure 5-1 (Part 2 of 2). Storage and Alignment Requirements of Scalar Data

Footnotes

¹ This is the default alignment boundary.

Note: For a table of ceil values, see Figure 9-4 on page 9-9.

Array Mapping

The alignment (a'), displacement (d'), and length (l') of an array element can be calculated using the rules given in the preceding section for an array of scalars, or in the following section for an array of structures. In the following equations, N is the number of array elements.

DATA MAPPING

The padding required between an array element ending at $d' + l'$ and the next element starting at d' is

$$\text{pad} = \text{mod}(-l', a')$$

where mod gives the smallest non-negative remainder after division.

No padding is needed after the last element. The total length of the array is therefore

$$l = N * l' + (N - 1) * \text{pad}$$

The alignment boundary and displacement of the array are those of the element:

$$\begin{aligned} a &= a' \\ d &= d' \end{aligned}$$

For example:

```
DECLARE 1 A (2,5),
        5 B BINARY (15),
        5 C BINARY (31);
```

A is an array of structures. According to the rules for mapping a structure, which are given in the next section, each element of A has a length of 6 bytes and starts at a displacement of 2 bytes from a word boundary:

$$a' = 4, \quad d' = 2, \quad l' = 6$$

The number of array elements (N) is ten. The padding between successive elements is:

$$\text{pad} = \text{mod}(-6, 4) = 2$$

The array is therefore mapped as:

$$\begin{aligned} a &= 4 \\ d &= 2 \\ l &= 10 * 6 + 9 * 2 = 78 \end{aligned}$$

Structure Mapping

The rules for structure mapping are:

1. Map the first immediate component, resulting in a , d , and l .
2. Given that the structure consisting of the first k immediate components has been mapped into a , d , and l (initially $k = 1$, and a , d , and l are the values obtained from step 1), map the immediate component $k + 1$, to obtain a' , d' and l' . Combine the two units therefore obtained according to the rules given below, to obtain a , d , and l for the structure consisting of the first $k + 1$ components.
3. Repeat step 2 with increasing values of k until the whole structure has been mapped.

This process is recursive if any of the components of the structure you are mapping is a structure.

Rules for Combining Two Units

To combine a unit mapped as a_1 , d_1 , and l_1 and a unit mapped as a_2 , d_2 , and l_2 into a new unit, which is mapped as a , d , and l , proceed as follows:

1. Some padding may be required after the first unit, which ends at d_1+l_1 , to align the second unit at d_2 :

$$\text{pad} = \text{mod}(d_2 - (d_1 + l_1), \min(a_1, a_2))$$

(This is calculated relative to $\min(a_1, a_2)$, which is the weaker of the two alignment boundaries. The proper boundary is taken into account in the calculation of a and d below.) Again, mod gives the smallest non-negative remainder after division. The offset of the second unit from the start of the new unit will be

$$l_1 + \text{pad}$$

2. The length of the new unit will be

$$l = l_1 + \text{pad} + l_2$$

3. Compare the alignment boundaries of the two units. If $a_2 > a_1$ (that is, if the alignment of the second unit is stronger than that of the first), the alignment boundary of the new unit is that of the second, and the displacement of the new unit must be calculated from that boundary:

$$a = a_2$$

$$d = \text{mod}(d_2 - (l_1 + \text{pad}), a_2)$$

If $a_2 \leq a_1$, the alignment boundary and displacement of the new unit are those of the first unit:

$$a = a_1$$

$$d = d_1$$

Example of Structure Mapping

This example shows the application of the structure mapping rules for a structure declared as follows:

```

DECLARE 1 A,
    5 B,
        10 C CHARACTER (2),
        10 D FLOAT DECIMAL (8),
        10 E BIT (2) ALIGNED,
        10 F CHARACTER (4),
    5 G,
        10 H BINARY (4),
        10 I PICTURE 'V99',
        10 J CHARACTER (1),
        10 K FLOAT DECIMAL (1);

```

Figure 5-2 and Figure 5-3 on page 5-14 show the steps involved in mapping the minor structures B and G, respectively. Figure 5-4 on page 5-15 shows how the results of mapping the two substructures B and G are combined to map the major structure A. The storage layout of structure A is shown in Figure 5-5 on page 5-15.

DATA MAPPING

| Name | a | d | l | pad | offset |
|-------|---|---|----|-----|--------|
| C | 1 | 0 | 2 | | |
| D | 8 | 0 | 8 | 0 | 2 |
| C,D | 8 | 6 | 10 | | |
| E | 1 | 0 | 1 | 0 | 10 |
| C,D,E | 8 | 6 | 11 | | |
| F | 1 | 0 | 4 | 0 | 11 |
| B | 8 | 6 | 15 | | |

Figure 5-2. Mapping Minor Structure B

As an example of applying the rules for combining two units, consider the step of combining units C and D in Figure 5-2. C and D have been mapped as

$$a1 = 1, \quad d1 = 0, \quad l1 = 2$$

and

$$a2 = 8, \quad d2 = 0, \quad l2 = 8$$

respectively (these values are obtained from Figure 5-1 on page 5-10). No padding is required:

$$pad = \text{mod}(d2 - (d1 + l1), a1) = \text{mod}(0 - 2, 1) = 0$$

D has the stronger boundary ($a2 = 8$), so

$$a = a2 = 8$$

and

$$d = \text{mod}(d2 - (l1 + pad), a2) = \text{mod}(0 - 2, 8) = 6$$

and

$$l = l1 + pad + l2 = 2 + 0 + 8 = 10$$

The unit obtained, which consists of fields C and D, will be used as the first unit in the next step.

| Name | a | d | l | pad | offset |
|-------|---|---|----|-----|--------|
| H | 2 | 0 | 2 | | |
| I | 1 | 0 | 2 | 0 | 2 |
| H,I | 2 | 0 | 4 | | |
| J | 1 | 0 | 1 | 0 | 4 |
| H,I,J | 2 | 0 | 5 | | |
| K | 4 | 0 | 4 | 1 | 6 |
| G | 4 | 2 | 10 | | |

Figure 5-3. Mapping Minor Structure G

As a second example, consider the step of combining the unit consisting of the fields H, I, and J with unit K, as shown in Figure 5-3, where

$a1 = 2, d1 = 0, l1 = 5$

and

$a2 = 4, d2 = 0, l2 = 4$

The padding between the two units is

$pad = \text{mod}(d2 - (d1 + l1), a1) = \text{mod}(0 - 5, 2) = 1$

Again, the second unit has the stronger boundary ($a2 = 4$), so

$a = a2 = 4$

and

$d = \text{mod}(d2 - (l1 + pad), a2) = \text{mod}(0 - 6, 4) = 2$

and

$l = l1 + pad + l2 = 5 + 1 + 4 = 10$

The resulting unit, G, is used as the second unit in Figure 5-4.

| Name | a | d | l | pad | offset |
|------|---|---|----|-----|--------|
| B | 8 | 6 | 15 | | |
| G | 4 | 2 | 10 | 1 | 16 |
| A | 8 | 6 | 26 | | |

Figure 5-4. Mapping Major Structure A

Figure 5-5 shows the resulting storage layout of structure A.

| Name of Item | Alignment Requirement | Length | Padding after Item | Offset from A (in bytes) | Displacement from Doubleword |
|--------------|-----------------------|---------|--------------------|--------------------------|------------------------------|
| C | byte | 2 bytes | | 0 | 6 |
| D | doubleword | 8 bytes | | 2 | 0 |
| E | byte | 2 bits | 6 bits | 10 | 0 |
| F | byte | 4 bytes | 1 byte | 11 | 1 |
| H | halfword | 2 bytes | | 16 | 6 |
| I | byte | 2 bytes | | 18 | 0 |
| J | byte | 1 byte | 1 byte | 20 | 2 |
| K | word | 4 bytes | | 22 | 4 |

Figure 5-5. Storage Layout of Structure A

STORAGE CONTROL

The following describes how to control the allocation of storage.

Variables of both problem data and program control data require storage. The attributes specified for a variable define the amount of storage required and how it is interpreted. For example:

STORAGE CONTROL

```
DECLARE INTEGER1 FIXED BINARY (31) AUTOMATIC;
```

A reference to `INTEGER1` is a reference to a fullword that contains a value to be interpreted as a fixed-point binary integer (see “Data Mapping” on page 5-9).

Storage allocation is the association of an area of storage with a variable, so that the data item to be represented by the variable can be recorded internally. When storage has been associated with a variable, the variable is said to be **allocated**.

The declaration of a variable includes a storage class attribute either by explicit specification or by default.

The way storage is allocated for a variable, and the degree of control you can exercise over storage, are determined by the storage class of that variable. There are three storage classes: static, automatic, and based. Each is specified by its corresponding storage class attribute.

You can specify the storage class for level-one variables only. Elements of arrays and members of structures inherit the storage class of the array or structure.

You cannot specify a storage class for a parameter or a named constant.

The default storage class attribute is `AUTOMATIC` for internal variables and `STATIC` for external variables.

Automatic and based variables can have internal scope only. Static variables can have either internal or external scope.

Using the `STATIC` Attribute

The allocation of storage for a static variable depends on the scope of the variable:

- If the variable has the `INTERNAL` attribute, storage is allocated on the first entry to the external procedure that contains the declaration.
- If the variable has the `EXTERNAL` attribute, storage is allocated on the first entry to the first external procedure that contains the declaration.

Storage for a static variable remains allocated until the run unit ends.

When storage is allocated, it is not initialized with zeros or blanks; the program must explicitly assign any initial values either by assignment statements or by use of the `INITIAL` attribute.

Variables declared with the `STATIC` attribute follow normal scope rules for the validity of references to them. For example:


```

MAINPROC: PROCEDURE OPTIONS (MAIN);
      .
      .
      .
      SUBPROC: PROCEDURE;
        DECLARE INTEGER1 FIXED STATIC INTERNAL;
        .
        .
        .
      END SUBPROC;
END MAINPROC;

```

In this example, although the variable `INTEGER1` is allocated throughout the program, it can be referenced by the name `INTEGER1` only within `SUBPROC` or any block contained in `SUBPROC`.

Using the INITIAL Attribute

The `INITIAL` attribute can only be used with the `STATIC` attribute. It specifies values assigned to a scalar or array variable when storage is allocated to it.

For a description of the syntax of the `INITIAL` attribute, see “`INITIAL` Attribute” on page 12-42.

You can specify the `INITIAL` attribute for arrays that do not have inherited dimensions, as well as for scalar variables. In a structure declaration, you can specify the `INITIAL` attribute only for field names.

You can specify only one initial value for a scalar variable, but more than one for an array variable. A structure variable requires separate initialization of each of its field names, if they are scalar or array variables.

The initial values specified for an array are assigned to successive elements of the array in row-major order, with the final subscript varying most rapidly.

If you specify fewer initial values than there are array elements, the remainder of the array is not initialized.

In the `INITIAL` attribute, you can specify only character, bit, or arithmetic constants (such as `'ABC'`, `'101'B`, or `3`) or the `NULL` built-in function (to initialize a static pointer variable).

For an array, the iteration factor specifies the number of times the item is to be repeated in the initialization of elements of the array.

If the attributes of any item in the `INITIAL` attribute differ from the attributes of the variable being initialized, conversion is processed.

The `INITIAL` attribute for a static external variable is ignored in all but the external procedure that allocated storage for the variable.

STORAGE CONTROL

Consider the following examples:

```
DECLARE NAME CHARACTER (10) STATIC
          INITIAL ('JOHN DOE');

DECLARE PI FIXED DECIMAL (5,4) STATIC
          INITIAL (3.1416);
```

When storage is allocated for NAME, it is initialized with the character string 'JOHN DOE' (padded on the right, with blanks, to 10 characters). When PI is allocated, it is initialized to the value 3.1416. The value can be retained throughout the program or changed while running.

Other examples are:

```
DECLARE SWITCH BIT (1) STATIC
          INITIAL ('1'B);

DECLARE MAXVALUE FIXED DECIMAL (2)
          STATIC INITIAL (99),
          MINVALUE FIXED DECIMAL (2)
          STATIC INITIAL (-99);
```

Consider the following example:

```
DECLARE A(15) CHARACTER(13) STATIC INITIAL
          ('JOHN DOE',
          'RICHARD ROW',
          'MARY SMITH'),
          B(10,10) DECIMAL FIXED(5) STATIC
          INITIAL((25)0,(25)1,(50)0);
```

In this example, only the first, second, and third elements of array A are initialized; the rest of the array is uninitialized. The array B is fully initialized, with the first 25 elements initialized to 0, the next 25 to 1, and the last 50 to 0.

Using the AUTOMATIC Attribute

Storage for an automatic variable is allocated on entry to the block in which it has been declared and is released when the block ends. Therefore, it can be reallocated many times during the running of a program. For a description of the syntax of the AUTOMATIC attribute, see "AUTOMATIC Attribute" on page 12-41.

You control the allocation of storage for an automatic variable by means of the block structure of your program. For example:

```

PROC1: PROCEDURE;
.
.
CALL PROC2;
PROC2: PROCEDURE;
    DECLARE (INTEGER1,INTEGER2) FIXED AUTOMATIC;
.
.
END PROC2;
.
.
CALL PROC2;
.
.
END PROC1;

```

In this example, each time PROC2 is called, the variables INTEGER1 and INTEGER2 are allocated storage, and when PROC2 ends the storage is released. Consequently, the values they contained are lost. The storage that has been released is available for reallocation to other variables.

You can specify an array bound or string length for an automatic variable as an integer constant or as an unsubscripted reference to an automatic, static, or parameter integer variable. In the latter case, the amount of storage allocated is determined while the program is running. For example:

```

A: PROCEDURE;
    DECLARE N FIXED BINARY;
    GET EDIT(N) (F(5));
.
.
B: PROCEDURE;
    DECLARE STR CHARACTER(N);
.
.
END B;
END A;

```

In this example, the character string STR will have a length defined by the value of the variable N that existed when procedure B was called.

A string length or array bound for an automatic variable must not be specified by means of another automatic variable declared in the same block.

Using the BASED Attribute

Based variables provide attributes for data accessed by a pointer value, such as data located in a buffer.

You declare a based variable with the BASED attribute. For a description of the syntax of the BASED attribute, see "BASED Attribute" on page 12-42.

STORAGE CONTROL

A based variable is always used together with a pointer value, which is taken from either the pointer-qualifier or the declaration of the based variable. This is described under "Based Variable Reference and Pointer Qualification" on page 5-20.

An example of a declaration of a based variable is:

```
DECLARE PLAYERS(30) CHARACTER (20) BASED (POINTER1);
```

In this example, PLAYERS is a 1-dimensional array of character data, with 30 elements. Its purpose is to redefine the characteristics of storage occupied by data assigned to another character variable, which might be declared as:

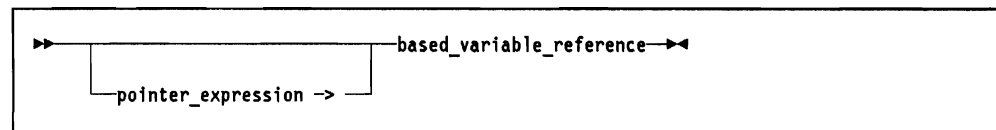
```
DECLARE TEAMS(2,15) CHARACTER (20);
```

After processing the assignment `POINTER1 = ADDR(TEAMS)`, the reference PLAYERS or `POINTER1 -> PLAYERS` refers to the same location in storage as the reference TEAMS, and `PLAYERS(20)` or `POINTER1 -> PLAYERS(20)` is equivalent to `TEAMS(2,5)`.

Based Variable Reference and Pointer Qualification

The name of a based variable, by itself, refers only to a collection of attributes. If it is to refer to a variable in storage, it must be associated with a pointer value, either explicitly or implicitly.

The syntax of a based variable reference is



pointer_expression

Is an expression of type pointer.

based_variable_reference

Is a reference to a based variable.

The pointer expression followed by the arrow is a **pointer qualifier**. If it is present, the variable reference is a **pointer-qualified reference**, and the association of the based variable with a pointer expression is called **pointer qualification**.

If no pointer qualifier is present, the based variable must have been declared with a pointer variable in the `BASED` attribute. The pointer variable is then implied as a pointer qualifier.

For example:

```
DECLARE INTEGER1    BINARY(15) BASED,  
        INTEGER2    BINARY(15) BASED (POINTER2),  
        POINTER1     POINTER,  
        POINTER2     POINTER;  
        . . .  
POINTER1->INTEGER1 = INTEGER2;
```

The reference to INTEGER1 is explicitly qualified by POINTER1. The reference to INTEGER2 is implicitly qualified by POINTER2.

When a reference to a based variable is evaluated, the explicit or implicit pointer qualifier is evaluated first, which gives a pointer value. This value must not be the null pointer and must not identify a location that has been released. The location of data in storage is identified by that pointer value, and the attributes of the data by those of the based variable.

Therefore, in the example above, POINTER1 -> INTEGER1 refers to the data accessed by POINTER1 and described by the attributes of INTEGER1, and INTEGER2 refers to the data accessed by POINTER2 and described by the attributes of INTEGER2.

Multiple Pointer Qualification

In a pointer-qualified reference, the pointer qualifier can be a variable reference, and can again be pointer-qualified. This leads to multiple pointer qualification. For example:

```

DECLARE P POINTER,
        Q POINTER BASED (P),
        R BINARY(15) BASED,
        QA POINTER,
        RA BINARY(15);
P = ADDR(QA);      /* P points to QA */
P->Q = ADDR(RA);  /* QA points to RA */
P->Q->R = 17;     /* RA = 17 */
    
```

In this example, P -> Q -> R is a reference to the based variable R with a pointer qualifier P -> Q, which is a reference to the based pointer variable Q with a pointer qualifier P. After the assignment to P, the references P -> Q and Q are equivalent to QA. After the assignment to Q, the references P -> Q -> R and Q -> R are equivalent to RA. By using implicit qualification, the assignments could be written as follows:

```

P = ADDR(QA);
Q = ADDR(RA);
Q->R = 17;
    
```

Pointer qualifiers can be subscripted and can be references to functions that return pointers. For example, given the declarations

```

DECLARE P(10) POINTER,
        Q ENTRY(CHARACTER(*)) RETURNS (POINTER) BASED,
        R BINARY (15) BASED;
    
```

a valid reference (with suitable declarations for A and I) is

```
P(I)->Q(A)->R
```

P(I) points to an entry value that accepts a character value as its argument and returns a pointer value. This entry is called with argument A. The returned pointer is used to access R. If Q is declared with no arguments:

```
DECLARE Q ENTRY() RETURNS(POINTER) BASED;
```

STORAGE CONTROL

the reference must have an empty argument list:

```
P->Q()->R
```

because Q refers to the entry value and not to the pointer returned by calling Q().

Relationship of Pointers and Based Variables

The data type of a based variable and the variable pointed to must match. This is true for scalars, and the elements of arrays and structures. According to the data mapping algorithm given under "Data Mapping" on page 5-9, this will be the case if:

- Both variables are scalar with identical data and alignment attributes, or the one is a character variable and the other a picture variable of equal length.
- Both variables are arrays with matching element descriptions. The number of dimensions or elements need not be the same, but the element or elements actually referred to must be contained in both descriptions.
- Both variables are structures whose attributes are identical up to the part referred to, including any minor structure containing the part.

In the examples of arrays given under "Using the BASED Attribute" on page 5-19, PLAYERS is used as a based variable and TEAMS as the variable pointed to.

In the following example:

```
DECLARE INTEGER1    FLOAT,  
        CHARSCALAR  CHARACTER (15),  
        CHARARRAY(15) CHARACTER (1) BASED (P);
```

a valid reference is:

```
ADDR(CHARSCALAR)->CHARARRAY
```

An example of structures is given in "Based Variables and Input/Output" on page 5-23.

ALLOCATE Statement for Based Variables

The ALLOCATE statement allocates storage for based variables and sets a pointer variable that can be used to identify the location, independent of procedure block boundaries.

The syntax of the ALLOCATE statement for based variables is:

```
▶▶—ALLOCATE—based_variable—SET(pointer_variable);—▶▶
```

Abbreviation: ALLOC for ALLOCATE

DATA ASSIGNMENT

```
DECLARE TRANSFILE      FILE RECORD INPUT,
      INRECORD        CHARACTER (18),
      POINTER1        POINTER;
DECLARE 1 DISPATCH BASED (POINTER1),
      5 RECORD_CODE  CHARACTER (1),
      5 PART_NO      CHARACTER (7),
      5 QUANTITY     CHARACTER (4),
      5 DEPT         CHARACTER (2),
      5 PROJECT      CHARACTER (4);
DECLARE 1 RECEIPT BASED (POINTER1),
      5 RECORD_CODE  CHARACTER (1),
      5 PART_NO      CHARACTER (7),
      5 QUANTITY     CHARACTER (4),
      5 SUPPLIER     CHARACTER (6);
DECLARE 1 DISPATCH_MSTRECORD,
      5 PART_NO      CHARACTER (7),
      5 QUANTITY     CHARACTER (4),
      5 DEPT         CHARACTER (2),
      5 PROJECT      CHARACTER (4);
DECLARE 1 RECEIPT_MSTRECORD,
      5 PART_NO      CHARACTER (7),
      5 QUANTITY     CHARACTER (4),
      5 SUPPLIER     CHARACTER (6);
      .
      .
      .
POINTER1 = ADDR(INRECORD);
READ FILE (TRANSFILE) INTO (INRECORD);
IF DISPATCH.RECORD_CODE = 'D'
  THEN
    DISPATCH_MSTRECORD
      = DISPATCH, BY NAME;
ELSE IF RECEIPT.RECORD_CODE = 'C'
  THEN
    RECEIPT_MSTRECORD
      = RECEIPT, BY NAME;
ELSE;
```

In this example, the record descriptions DISPATCH and RECEIPT are both associated with POINTER1, which holds the address of INRECORD. Once a record has been read into INRECORD, RECORD_CODE is tested to determine if a DISPATCH or RECEIPT record has been read. The appropriate fields are then assigned to DISPATCH_MSTRECORD or RECEIPT_MSTRECORD. If the record code for INRECORD is not valid, no action is processed.

Data Assignment

Conversions between any types of problem data are valid in assignments.

Assignments are:

- An assignment in an assignment statement
- The initialization of a variable in a declaration

- An assignment to the control variable of a do-group
- An assignment in a stream input or output operation
- An assignment to a dummy argument when passing the dummy argument to a procedure or built-in subroutine
- Returning a value from a function.

Conversions may also be processed in comparisons, so that the items being compared have the same format.

Before a value is assigned, it is converted to the type of the target. The precision or length of the target may differ from that of the converted source. The effect of differing lengths or precisions is explained in the following sections.

String Data Assignment

The source string is assigned to the target string. If the source string is longer than the target, excess characters or bits on the right are truncated. If the source string is shorter than the target, the value being assigned is padded on the right. Character values are padded with blanks, bit values with zeros.

Examples are:

```
DECLARE ACHAR CHARACTER(10);
ACHAR = 'TRANSFORMATIONS';
```

'TRANSFORMATIONS' has 15 characters. Therefore, five characters will be truncated from the right-hand end of the string when it is assigned to ACHAR. This is equivalent to

```
ACHAR = 'TRANSFORMA';
```

The following statements assign equivalent values to ACHAR:

```
ACHAR = 'PHYSICS';
ACHAR = 'PHYSICS  ';
```

The first assignment pads the character value on the right with three blank characters.

The following statements assign equivalent values to ABIT:

```
DECLARE ABIT BIT(10) ALIGNED;
ABIT = '110011'B;
ABIT = '1100110000'B;
```

The first assignment pads the bit value on the right with four zeros.

The following statements assign different values to ACHAR:

```
ACHAR = '110011'B;
ACHAR = '1100110000'B;
```

Each assignment converts the bit constant to a character value. The first pads the value on the right with four blank characters and is equivalent to:

DATA ASSIGNMENT

```
ACHAR = '110011  ';
```

The second is equivalent to:

```
ACHAR = '1100110000';
```

The source string must not identify a string that starts to the left of and overlaps the target string.

For example, the assignment:

```
SUBSTR(ACHAR,3,5) = SUBSTR(ACHAR,1,5);
```

is not valid, because the string identified by the SUBSTR built-in function starts to the left of the target string and overlaps it.

Arithmetic Data Assignment

A fixed-point target whose number of integer digits is smaller than that of the source may be too small to hold the source value, in which case the result is undefined.

For example:

```
DECLARE AFID FIXED DECIMAL (1) STATIC INITIAL (5),  
        BFID FIXED DECIMAL (3,2),  
        CFID FIXED DECIMAL (2,1);
```

```
BFID = (AFID * 5) / CFID;
```

In this example, the intermediate result of $AFID * 5$ is 25. The result of dividing 25 by the value of CFID fits the precision of the target BFID if the value of CFID is greater than 2.5, but loses significant digits if CFID is 2.5 or less. In both cases, the result may lose fractional digits.

A fixed-point decimal target whose scale factor is smaller than that of the source loses digits on the right, starting with the least significant digit. For example:

```
DECLARE AFID FIXED DECIMAL (5,3) STATIC  
        INITIAL (12.987),  
        BFID FIXED DECIMAL (5,1);  
BFID = AFID;
```

In this example, the value of AFID is assigned to the target BFID. Because the scale factor of BFID is smaller than that of AFID, the fractional part of the value is truncated (it loses the two least significant digits), giving 12.9. To round a value, rather than let it be truncated, use the ROUND built-in function.

If the number of integer digits of the source value is less than that of the target, the source value is padded with zeros on the left. Similarly, if the fractional part of the source value is smaller than that of the target, the source value is padded with zeros on the right. For example:

```
DECLARE AFID FIXED DECIMAL (3,1) STATIC  
        INITIAL (12.3),  
        BFID FIXED DECIMAL (5,2);  
BFID = AFID;
```

In this example, the value of AFID is assigned to BFID and padded on the right to the precision declared for BFID. The value of BFID is then represented as 012.30.

If a source value is assigned to a floating-point target that has a smaller precision, the value of the target may be less accurate than the original source value. For example:

```
DECLARE AFID FIXED DECIMAL (5) STATIC
          INITIAL (13579),
          BFLOD FLOAT DECIMAL (3);
BFLOD = AFID;
```

In this example, the value of AFID is converted to floating-point decimal with the maximum precision associated with short floating-point, and is assigned to BFLOD. However, the value of BFLOD is treated as having precision (3) whenever it requires conversion to a non-arithmetic data type. In that case, its value is represented as $1.36 * 10^{**4}$.

When you assign a scalar variable to a structure, the scalar is assigned to each field in the structure. Conversions are processed when necessary. Consider the following example:

```
DECLARE CHAR1 CHARACTER (1) STATIC INITIAL ('0');
DECLARE 1 SAMPLESTRUCTURE,
        5 CHARFIELD CHARACTER (6),
        5 BINFIELD  BINARY FIXED (15),
        5 BITFIELD  BIT (10) ALIGNED,
        5 PICFIELD  PICTURE '9';
```

The assignment statement

```
SAMPLESTRUCTURE = CHAR1;
```

would assign CHAR1 to CHARFIELD, with five blank characters of padding to the right. It would place the binary value zero in BINFIELD, the bit value zero in BITFIELD, and the EBCDIC character value '0' in PICFIELD.

Data Conversion

Data conversion may occur between any types of problem data, but may not occur between different types of program control data.

Problem data may be converted in the following circumstances:

- In an assignment (see "Data Assignment" on page 5-24 for the various types of assignment).
- When passing an argument to a built-in function.
- When using a conversion built-in function.
- When evaluating an operational expression.

Before a data item can be converted, all the data attributes of the target must be determined, including the length of a string target and the precision of an arithmetic target.

DATA CONVERSION

Built-In Conversion Functions

You can convert problem data from one type to another by means of one of the built-in conversion functions:

| | |
|-----------|---------|
| BINARY | DECIMAL |
| BIT | FIXED |
| CHARACTER | FLOAT |

These are described in Chapter 15, “Built-In Functions, Subroutines, and Pseudovariables.”

Each function returns a value with the attribute specified by the function name.

Before you use data as operands, you may have to convert them explicitly to the type required by the operator (see “Operational Expressions” on page 9-4). For example:

```
DECLARE ACHAR CHARACTER (8)
        STATIC INITIAL('01111110'),
        ABIT BIT(8) STATIC ALIGNED
        INITIAL('10000001'B),
        BBIT BIT(16) ALIGNED;
BBIT = BIT(ACHAR) || ABIT;
```

In this example, the two strings concatenated, ACHAR and ABIT, are a character string and a bit string. One operand must therefore be converted to the data type of the other. In this example, the BIT built-in function converts the character operand to a bit operand.

Similarly, for most arithmetic operators, if one operand is fixed-point decimal with a nonzero scale factor and the other is fixed-point binary, you must convert the binary operand to a decimal operand (see “Results of Arithmetic Operations” on page 9-5). For example:

```
DECLARE AFID FIXED DECIMAL(7,2),
        BFID FIXED DECIMAL(8,2),
        CFIB FIXED BINARY(15);
BFID = AFID + DECIMAL(CFIB);
```

In this example, AFID and CFIB are incompatible because AFID has a nonzero scale factor. CFIB must therefore be converted to decimal, which is done here by means of the DECIMAL built-in function.

Forcing a conversion may save processing time. When the types of arithmetic operands give a result with a base or scale different from those of the target, you can avoid the final conversion by converting one operand. For example:

```
DECLARE AFID FIXED DECIMAL(5),
        BFID FIXED DECIMAL(6),
        CFIB FIXED BINARY(15);
BFID = AFID + DECIMAL(CFIB);
```

In this example, the operands are compatible, as AFID has a scale factor of zero. Without the forced conversion of CFIB to decimal, AFID would be converted to

binary, and the result of the operation would be converted to decimal. By converting CFIB to decimal, one conversion is avoided.

The conversions done by the conversion built-in functions can also be done by assignment to a variable that has the required attributes. You may, however, find the built-in functions more convenient than creating a variable just for a conversion.

Calculating String Length and Precision

In the following cases, the target's length or precision is calculated first:

- When converting an operand before processing an arithmetic or comparison operation.
- When converting arguments of certain built-in functions before calling them.
- When converting a data item from arithmetic to string. For example, conversion from fixed-point decimal to bit involves an initial conversion to fixed-point binary.
- When converting an arithmetic or bit argument to character, before passing it to a parameter declared as CHARACTER(*), or when converting an arithmetic or character argument to bit, before passing it to a parameter declared as BIT(*) .

Conversion Rules

This section gives the rules for conversion and explains when and how the precision of a target is calculated. The rules are given according to the data attributes of the target and source.

Target: Arithmetic

SOURCE:

Arithmetic

Conversion is necessary only when the source and target differ in base or scale.

The target's precision is calculated when any of the cases given under "Calculating String Length and Precision" applies. The equations used in these conversions are given in Figure 12-4 on page 12-27.

| | BINARY FIXED source | DECIMAL FIXED or PICTURE source | BINARY FLOAT source | DECIMAL FLOAT source |
|---------------------------|------------------------|---|------------------------|-------------------------|
| BINARY FIXED target | | $p_2 = \min(\text{ceil}(p_1 * 3.32) + 1, 31)$ $q_2 = 0$ (see note 2) | (see note 3) | (see note 3) |

Figure 5-6 (Part 1 of 2). Equations for Calculating Converted Precision

DATA CONVERSION

| | BINARY FIXED source | DECIMAL FIXED or PICTURE source | BINARY FLOAT source | DECIMAL FLOAT source |
|----------------------------|--|---------------------------------------|-------------------------------|---|
| DECIMAL FIXED target | $p_2 = \text{ceil}(p_1/3.32) + 1$ $q_2 = 0$ | (see note 4) | (see note 3) | (see note 3) |
| BINARY FLOAT target | $p_2 = p_1$ | $p_2 = \text{ceil}(p_1 * 3.32)$ | | $p_2 = \text{min}(\text{ceil}(p_1 * 3.32), 53)$ |
| DECIMAL FLOAT target | $p_2 = \text{ceil}(p_1/3.32)$ | $p_2 = p_1$ | $p_2 = \text{ceil}(p_1/3.32)$ | |

Figure 5-6 (Part 2 of 2). Equations for Calculating Converted Precision

Note:

1. p_1 is the number of digits of the source, p_2 is the number of digits of the target, and q_2 is the target's scale factor.
2. $\text{min}(x,y)$ is the smaller of x and y .
3. For a table of ceil values, see Figure 9-4 on page 9-8.
4. The scale factor of the source must be zero. If the number of digits of the source is greater than 9, and the number of binary digits needed is greater than the maximum, 31, the result may be undefined.
5. If this conversion occurs, the target precision is always known.
6. If the source is picture, $p_2 = p_1$ and $q_2 = q_1$.

Conversion to a pictured target occurs only when data is assigned; in such cases, the target's precision is always known.

The equations in Figure 12-4 may also help you decide what precisions to give to variables. For example, the precision of a fixed-point decimal or pictured target in an assignment statement should be large enough to hold the maximum value of the source:

- For fixed-point binary (p_1), the equation in Figure 5-6 on page 5-29

$$p_2 = \text{ceil}(p_1/3.32) + 1$$
gives the minimum number of decimal digits needed to represent p_1 binary digits. If you want a target scale factor of q_2 , derive the number of digits, p_2 , from the formula

$$p_2 \geq \text{ceil}(p_1/3.32) + 1 + q_2$$
- For fixed-point decimal (p_1, q_1), derive the precision and scale factor (p_2, q_2) from the formulas

$$p_2 \geq p_1 - q_1 + q_2$$

$$q_2 \geq q_1$$
- For floating-point binary (p_1), derive the precision (p_2) from the formula

$$p_2 \geq \text{ceil}(p_1/3.32)$$

- For floating-point decimal (p_1), derive the precision (p_2) from the formula

$$p_2 \geq p_1$$

- For a floating-point source, the exponent must be small enough for the source value to fit the target.

The accuracy of a value may be affected when converting between binary and decimal representations. These conversions use the factor 3.32 when calculating the target precision.

CHARACTER

Conversion from character to arithmetic is valid only in an assignment or when using the BINARY, DECIMAL, FIXED, or FLOAT built-in functions.

The source character value must represent an optionally signed arithmetic constant. It may be surrounded by blanks, but no blanks may appear between the sign and the constant. A null string or a string of blanks is permitted and is interpreted as a value of zero.

If the source string does not satisfy these conditions, the conversion condition is raised.

The numeric value represented by the optionally signed constant is converted to the attributes of the target, using the attributes of the constant as source attributes.

BIT

Conversion from bit to arithmetic is valid only in an assignment or when using the BINARY, DECIMAL, FIXED, or FLOAT built-in functions.

The source bit value is interpreted as an unsigned binary integer and is converted to the attributes of the target, using FIXED BINARY as the source attributes. A null bit value is interpreted as a value of zero.

Target: Character

Conversion to character is valid only in an assignment or when using the CHARACTER built-in function.

SOURCE:

Coded Arithmetic

The number of digits of an arithmetic value that is converted to a character value must be greater than or equal to the scale factor.

The coded arithmetic value is represented as a decimal constant, preceded by a minus sign if it is negative, as described below. The constant is converted to an intermediate character result whose length is derived from the attributes of the source. The intermediate result is assigned to the target according to the rules for string assignment.

FIXED BINARY (p_1)

The binary precision (p_1) is first converted to the equivalent decimal precision ($p_2, 0$), where

DATA CONVERSION

$$p_2 = \text{ceil}(p_1/3.32) + 1$$

Thereafter, the rules are the same as for FIXED DECIMAL to CHARACTER.

FIXED DECIMAL (p_1, q_1)

Conversion is done in the following way:

1. The constant is right adjusted in a field whose width is $p_1 + 1$ (The three added bytes allow for a minus sign, a decimal point, and a leading zero before the point.)
2. Leading zeros to the left of the decimal point are replaced by blanks, except for the rightmost zero if it immediately precedes the decimal point.
3. A minus sign precedes the first digit of a negative value. A positive value is unsigned.
4. If $q_1 > 0$, a decimal point appears and the constant has q_1 fractional digits; if $q_1 = 0$, no decimal point appears.

The following examples show the intermediate strings generated from fixed-point decimal values. The letter b indicates a blank character.

| Precision | Value | String |
|-----------|--------|------------|
| (5,0) | 2947 | 'bbbb2947' |
| (4,1) | -121.7 | 'b-121.7' |
| (4,4) | -0.2 | '-0.2000' |

FLOAT BINARY (p_1)

The floating-point binary precision (p_1) is first converted to the equivalent floating-point decimal precision (p_2), where $p_2 = \text{ceil}(p_1/3.32)$. Thereafter, the rules are the same as for FLOAT DECIMAL to CHARACTER.

FLOAT DECIMAL (p_1)

A floating-point decimal source is converted as if it were transmitted by an E-format item of the form:

E(w,d)

where:

w

is the length of the intermediate string, $p_1 + 7$.

d

is the number of fractional digits, $p_1 - 1$.

The following examples show the intermediate strings generated from floating-point decimal values:

| Precision | Value | String |
|-----------|---------------|----------------|
| (5) | 1735 * 10**5 | 'b1.7350E+08b' |
| (5) | -.001663 | '-1.6630E-03b' |
| (3) | 1 | 'b1.00E+00b' |
| (1) | 0 | 'b0.E+00b' |
| (3) | 25 * 10**-101 | 'b2.50E-100' |

PICTURE

The character value of the pictured field is assigned to the target string according to the rules for string assignment.

BIT

'0'B becomes the character 0 and '1'B becomes the character 1. The null bit string becomes the null character string. The obtained character value is assigned to the target string according to the rules for string assignment.

Target: Bit

Conversion to bit is valid only in an assignment or when using the BIT built-in function.

SOURCE:

Arithmetic

If necessary, the arithmetic value is converted to binary fixed-point, and both the sign and any fractional part are ignored. The resulting binary integer is treated as a bit value. It is assigned to the target according to the rules for string assignment.

FIXED BINARY (p₁)

The result is a bit string of length p₁. If p₁ is zero, the result is the null bit string.

The following examples show the intermediate strings generated from fixed-point binary values:

| Precision | Value | String |
|-----------|-------|--------|
| (1) | 1 | '1'B |
| (3) | -3 | '011'B |

FIXED DECIMAL (p₁,q₁) and PICTURE

The fixed-point decimal value of a pictured field is used; its precision is taken from the corresponding picture specification.

The length of the intermediate bit value is given by

$$\min(\text{ceil}((p_1 - q_1) * 3.32), 31)$$

Only the number of integer digits of the source is used in this conversion; any fractional digits are ignored. The number of integer digits is p₁-q₁. (p₁ = q₁ results in a null bit string.)

DATA CONVERSION

The following examples show the intermediate strings generated from fixed-point decimal values:

| Precision | Value | String |
|-----------|-------|---------|
| (1) | 2 | '0010'B |
| (2,1) | 2.2 | '0010'B |

Fractional digits are lost.

FLOAT BINARY (p_1)

The length of the intermediate bit value is given by

$\min(31, p_1)$

FLOAT DECIMAL (p_1)

The length of the intermediate bit value is given by

$\min(\text{ceil}(p_1 * 3.32), 31)$

CHARACTER

Character 0 becomes '0'B and character 1 becomes '1'B. Any character other than 0 or 1 raises the conversion condition. The null character string becomes the null bit string. The generated bit value, which has the same length as the source character value, is assigned to the target according to the rules for string assignment.

Truncation of Floating-Point Data

If a PL/I statement involves conversion of a floating-point data item in which the digits in the original copy of the data item will not all fit into the converted copy, then the rightmost excess digits are truncated. Such truncation can occur when converting floating-point data items from long form to short form, or from floating-point to fixed-point or picture data.

In some kinds of conversions, such as from floating-point to character, an intermediate conversion from floating-point to fixed-point is done, followed by a final conversion from fixed-point to character. Truncation can occur during the intermediate conversion.

Examples of Data Conversion

The following example shows the values obtained when assigning decimal integer and bit constants to bit variables.

```
DECLARE ABIT BIT (1) ALIGNED,  
        BBIT BIT (5) ALIGNED;  
ABIT=1; /*ABIT HAS VALUE '0'B*/  
BBIT=1; /*BBIT HAS VALUE '00010'B*/  
BBIT='1'B; /*BBIT HAS VALUE '10000'B*/
```

The assignment to ABIT results in the following sequence of actions:

1. The decimal constant 1 has the attributes FIXED DECIMAL (1,0) and is converted to a fixed-point binary value of precision (4), following the rules for con-

version from fixed-point decimal to fixed-point binary. The value is now 1, represented as a 4-digit binary number.

2. This value is converted to a bit value of length 4 and becomes '0001'B.
3. The bit value is assigned to ABIT. Because the length of ABIT is 1 and the length of the bit value assigned is 4, the value is truncated on the right. ABIT has a final value of '0'B.

In the first assignment to BBIT, the sequence of actions is similar to that described for ABIT, except that the value is extended at the right with a zero, because the length of BBIT is one greater than that of the value to be assigned.

The following example shows the values obtained when assigning character and integer constants to character variables:

```
DECLARE ACHAR CHARACTER (4),
        BCHAR CHARACTER (7);
ACHAR='0'; /*ACHAR HAS VALUE '0  '*/
ACHAR=0;  /*ACHAR HAS VALUE '  0'*/
BCHAR=1234567; /*BCHAR HAS VALUE ' 1234'*/
```

In the first assignment, the character constant '0' is assigned to ACHAR and padded on the right with blanks.

In the second assignment, the integer constant 0 has the attributes FIXED DECIMAL(1,0). It is converted to a character value, which introduces three blanks on the left, and assigned to ACHAR.

In the third assignment, the integer constant is converted to the character value 1234567. This value is truncated on the right and assigned to BCHAR.

DATA CONVERSION



Chapter 6. AS/400 PL/I File and Record Management

This chapter gives an overview of the concepts involved in using AS/400 files.

Topics include:

- File management
- Types of files
- Using record formats.

Detailed information about PL/I input/output is given in Chapter 8, "Using AS/400 Files." This includes specific information about commitment control and the %INCLUDE directive. Complete information on AS/400 files can be found in the *Programming: Data Management Guide*.

File Management

Files and their use are controlled by OS/400. This section describes file independence, device independence, system override considerations, security, and authority.

File Independence

The key element for all input/output operations on the AS/400 System is the file. All files used on the system are defined to OS/400. OS/400 maintains a description of each file that is accessed by a program when the file is used.

The files are online and serve as the connecting link between a program and the data. The actual device association is made when the file is processed. In some instances, this type of input/output control allows the user to change the attributes of the file used in a program without changing the program.

PL/I files and AS/400 files are associated either by default, by the use of the TITLE option of the OPEN statement, or by the use of a CL override command, such as OVRTPEF or OVRPRTF. For further information on the CL override commands, see the *Programming: Control Language Reference*.

There are a few cases of file dependence. For example, the INTERACTIVE option of the ENVIRONMENT attribute usually implies a display file, and the INDEXED option implies a keyed data base file.

Device Independence

PL/I is, to a large extent, device independent. For example, if you specify the CONSECUTIVE option of the ENVIRONMENT attribute, and do not use the OPTIONS option of the input/output statements, you can associate your PL/I file with any AS/400 device.

Spooling

The AS/400 System provides for the use of input and output spooling functions. Each AS/400 printer and diskette description contains a spool attribute that determines if spooling is used for the file at run time. The PL/I program is not aware that spooling is being used. The actual physical device from which a file is read or to which a file is written is determined by the spool reader or the spool writer. See the *Programming: Data Management Guide* for more information on spooling.

Output Spooling

Output spooling is valid for batch and interactive jobs.

File override commands can be used at run time to override the spooling options that are specified in the file description, such as the number of copies printed. In addition, AS/400 spooling support allows you to redirect a file after the program has run. For example, you can direct the printer output to a different device, such as diskette.

Input Spooling

Input spooling is valid only for inline data files in batch jobs. If the input data read by PL/I comes from a spooled file, PL/I is not aware of which device the data was spooled in from. See the *Programming: Data Management Guide* for more information on inline files.

System Override Considerations

CL system override commands, such as OVRTPEF or OVRPRTF, may be used for several reasons:

- To change or add file attributes.
- To redirect a PL/I file to a different AS/400 file at run time.

The name you specify on the FILE option of the override command is used to match the file name in either the file declaration or the TITLE option of the OPEN statement. For a discussion of how to use system override commands, refer to the *Programming: Data Management Guide*.

If you issue the CL command OVRDBF (Override Data Base File) for a PL/I file specifying a value for the MBR parameter, any member name you specify using TITLE in the OPEN statement for the file is ignored. Similarly, a CL override command for a PL/I file may override some of the ENVIRONMENT attribute options specified for the file. An example of an option that may be overridden that is also directly specifiable within the PL/I program is *EXCL or *EXCLRD lock state on data base files.

Before a file can be accessed, you may need to specify additional file attributes on either an override command or a create or change system object command.

Overrides may be used to direct PL/I input and output to a different file than the program intended. For example, a PL/I file with the attributes of SEQUENTIAL, INPUT, and CONSECUTIVE could be used with a data base file, a tape file, a diskette file, or an inline file.

When file redirection is used, you must ensure that the program is not attempting any I/O statement options that are not defined for the file type. For example, if the program is using POSITION(PREVIOUS) on a READ statement to a data base file, the file in the program cannot be directed to a diskette file.

In general, input/output statement options not allowed for the file type will result in the ERROR condition being raised. The exceptions to these are the RECORD and INDICATORS options which are ignored by file types that do not support these options. For example, the RECORD option is ignored by diskette files, and the INDICATORS option is ignored by physical and logical data base files.

Security

Authority and ownership apply to a data base file on a file level. That is, if a user profile has a certain authority to a data base file, then that user may do the authorized functions on any and all members of the file.

Ownership

Ownership of a file and authorization to it interact in the following ways:

- An owner of a file always gets all rights to the file.
- An owner's rights can be revoked; but even when the owner has no explicit rights to the file, the owner may process GRTOBJAUT and RVKOBJAUT commands on it.
- Changing the owner does not revoke the old owner's rights to the file.

Authority

There are three Object Rights for a file: *OBJOPR, *OBJEXIST, and *OBJMGT. The four Data Rights for a file are: *READ, *ADD, *UPD, and *DLT. *CHANGE authority for a logical file is defined as *OBJOPR and for a physical file it is *OBJOPR plus all of the data rights.

The following rules for data rights allow the user to protect a data base:

- Data rights apply only to a physical file. An attempt to grant data rights to a logical file is ignored.
- A file cannot be opened without *OBJOPR right to the target file (either physical or logical). After the file is open, an I/O operation will not work without the corresponding data right on the physical file that contains the data.
- The AUT parameter allows you to specify what authority the public has for the program. You can use AUT to specify a number of different types of protection.

Types of Files

The following AS/400 files are supported by AS/400 PL/I:

- Data Base Files
- Display Files
- Diskette Files
- Tape Files
- Printer Files
- Data File Management (DDM) Files.

The following AS/400 files are supported by System/38 Environment only:

- Communications Files
- BSC Files
- Inline Files.

The following sections introduce each type. For more information, see the *Programming: Data Management Guide*.

Data Base Files

A data base file is a collection of records. The sequence of these records and the fields contained in a record are a part of the system file definition. The fact that the field definition is part of the file definition is one of the features of the AS/400 architecture. A physical file stores records in the same format and/or sequence in which you can access them. A logical file allows you to specify a description of how the records in a physical file may be accessed.

Physical Files

A physical file contains fixed-length records with one record format. The records are physically stored in arrival sequence. You may access these records in either a specified keyed sequence or arrival sequence.

Logical Files

A logical file does not contain records, but contains a set of instructions that the system uses to access a physical file. A logical file allows you to define different views of the fields in the records in a physical file, and to define different sequences for accessing these records.

File Organization

The AS/400 data base file organization is based on arrival sequence or keyed sequence access paths.

Access Paths

The access path determines the order in which the data records are returned to the program when a file is accessed.

Arrival Sequence Access Path: An arrival sequence access path is based on the order in which the records are stored in the file. For retrieval or updating, records can be accessed in one of the following ways:

- Sequentially, where each record is taken from the next sequential position in the file.
- Directly by relative record number, where the record is identified by its position from the beginning of the file.

An arrival sequence access path is valid only for the following:

- A member of a physical file, where no key fields are specified.
- A logical file in which each member of the logical file is based on only one member of only one physical file, where no key fields are specified.

Keyed Sequence Access Path: For a keyed sequence access path, the sequence of the records in a file is based on the contents of the key fields as defined in the DDS (Data Description Specifications). This type of access path is updated whenever records are added, deleted, or changed.

For retrieval or updating, records can be accessed in one of the following ways:

- Sequentially, where each record is taken from the next sequential position in the file.
- Directly by relative record number, where the record is identified by its position from the beginning of the file.
- Directly by means of the key fields as defined in the DDS.

The keyed sequence access path is valid for any physical or logical data base file type. The sequencing of records in the file is defined in the DDS for the file when the file is created and is maintained automatically.

Record Formats

A record format definition is a list of names and attributes of fields in the order in which they should appear in a record. This definition is made through DDS, and includes the field name, the data type (binary, packed decimal, floating-point, zoned decimal, or character), and the length of the field. Files use record formats in the following ways:

- A physical file has one record format. This record format determines the actual storage attributes and order of the fields for the records in the file.
- A logical file associates record formats with based-on physical file(s) and member(s). A logical record format defines a logical view of records in the physical file. The fields in a physical record format map to the fields in the logical record format.

TYPES OF FILES

Members

In the discussion so far, a data base file has been considered a means of accessing a collection of records. It is more accurate to say that a collection of records is accessed with a member of the file.

The ability to have many members within a file allows you to describe data characteristics once at the file level, then operationally separate collections of records created, each having these same characteristics. Therefore:

- A file may be defined to allow any number of members.
- Members can be added to and deleted from a file.
- Each physical member consists of a separate stored collection of records with an independent arrival sequence.
- Each keyed member consists of a separate access path over the data in physical file members. Therefore maintenance is done on a member basis.

If a file is defined with a maximum of one member, the fact that the file actually has a member makes no difference to the user. This is the case because the first member of a file is the default member used in operations when no member name is specified.

If a file is overridden with the CL command OVRDBF specifying MBR(*ALL), all the members in the file are processed, one at a time, in the order they were originally created. After the last member is processed, an ENDFILE condition is raised. If you specify the POSITION parameter of the READ statement with NXTUNQ, PRVUNQ, NXTEQL, PRVEQL, FIRST, or LAST, the search for the record will be made only within the member currently being accessed.

File Locking

File locking is used to control the access to physical file members. If more than one job is running at the same time, and all are accessing the same physical file members, some degree of file locking may be necessary in order to synchronize the use of the data.

There are five different lock states that may be applied to the records in a physical file member. When a particular lock is specified, all of the data records in each physical file member associated with the member being opened are locked in the specified state. For logical file members, this means that all of the data records in all of the base physical file members are locked in the specified state. The file lock states are:

- Exclusive (*EXCL)
- Exclusive Allow Read (*EXCLRD)
- Shared for Update (*SHRUPD)
- Shared No Update (*SHRNUP)
- Shared for Read (*SHRRD).

For a definition of these lock states, refer to the *Programming: Data Management Guide*.

You may specify or imply all but the *SHRNUP lock state from within a PL/I program. *EXCL or *EXCLRD may be specified as options on the ENVIRONMENT attribute. *SHRUPD is the default for a file with the UPDATE or OUTPUT attribute specified, and no lock state specified in the ENVIRONMENT attribute. *SHRRD is the default for a file with the INPUT attribute specified, and no lock state specified on the ENVIRONMENT attribute.

Notes:

1. File locks are not checked from within one running job. File locks only prevent access from other concurrently running jobs. This means that from within one running job, a file with *EXCLRD may be opened more than once.
2. The CL command OVRDBF (Override Data Base File) may override the lock state specified or implied by the PL/I program.

Record Locking

Record locking is used to control the updating of records within data base files. A record lock is similar to an *EXCLRD lock on a file. The user holding the record lock has exclusive update rights to the record, but other users may read the record. (Reading a locked record under commitment control may not be allowed. See "Using the PLICOMMIT Built-In Subroutine" on page 8-59.)

Locks apply only to records in physical files. A record in a logical file is always mapped to a particular record in a physical file. Locks are maintained on a physical record by each logical or physical file. If a record is accessed by two different files, the second file to access the record will find the record already locked. Because only physical records are locked, there may be many records in different logical files that all map to the same physical record. You must be aware of which physical files are used by the various logical files and ensure that all files are used in a consistent manner.

When a READ statement is processed for a file with the UPDATE attribute, the record just read will be locked. If the record is already locked to another concurrently running job, the READ statement will wait until either the record lock is released or a record lock wait timeout occurs. If a record lock wait timeout occurs, the TRANSMIT condition is raised. The record lock wait time defaults to 60 seconds but may be set by the user through the create file or override commands.

A record lock is normally released whenever another READ, REWRITE, or DELETE statement is directed to a different record in the same open PL/I file. If another READ statement is processed to the same record in the same open PL/I file, the record lock is not released. If a REWRITE or DELETE is directed to a record just read for UPDATE in the same open PL/I file, the record lock is released after the REWRITE or DELETE statement is processed.

The record lock held after a READ statement will be released if a subsequent READ statement ends with a KEY condition. You may use this fact to release a record lock by forcing a KEY condition to occur. If a record is locked to a file and you do not release the lock through the use of a subsequent READ, REWRITE, or DELETE statement, the record will remain locked to the file until either you issue a CLOSE statement, or the file is closed by PL/I at the end of the main procedure.

TYPES OF FILES

A REWRITE or DELETE statement with the KEY option attempts to obtain a record lock prior to the REWRITE or DELETE operation. This means that a REWRITE or DELETE statement may wait if the record is already locked. If the record lock timeout occurs, the REWRITE or DELETE statement will end with TRANSMIT condition.

Note that unlike file locks, record locks are checked within the same running job. This means that it is possible to receive a record locked condition on a record that is locked within the same job. This can occur if the same physical file record is accessed for UPDATE through different PL/I files. If this occurs, the READ statement ends immediately with TRANSMIT condition. The record wait time is not used in this case.

If commitment control is active for a file, the releasing of record locks is delayed until a call to PLICOMMIT or PLIROLLBACK is processed. For the rules on record locks under commitment control, see "Record Locks" on page 8-62.

DEVICE Files

A device file is a description of how input data is presented to a program from a device, or how output data is presented to the device from a program. It is not necessary to have a separate device file for each device; you can use only a single device file for several different devices of the same class by using an override command. Any number of device files can be associated with one device, but only one device description can exist for each device. For more information on device descriptions, see the *Programming: Data Management Guide*.

The types of device files supported by PL/I are:

- Display Device Files
- Diskette Files
- Tape Files
- Printer Files
- DDM Files.

The types of device files supported by PL/I in System/38 Environment only are:

- Communications Files
- BSC Files.

DISPLAY Files

Display device files are used to format the display. They describe input and output fields, constants, the use of command function and command attention keys, and the handling of errors. Display files can be program-described files or externally described files.

Subfiles

A subfile is a group of identically formatted records that is read from or written to a display device file. Subfiles can be specified in the DDS for a display device file to allow the user to handle multiple records of the same type on the display. For example, a program reads records from a data base file and creates a subfile of output records. When the entire subfile has been written, the program sends the entire subfile to the display device in one write operation. The user can change data or enter additional data in the subfile. The program then reads the entire subfile from the display device into the program and processes each record in the subfile individually.

Descriptions of records included in a subfile are specified in the DDS for the file. The number of records that can be contained in a subfile must also be specified in the DDS. One file can contain more than one subfile, and up to twelve subfiles can be active and displayed concurrently.

The DDS for a subfile consists of two record formats: a subfile record format and a subfile control record format. The subfile record format contains the field descriptions of all the records in the subfile. Specification of the subfile control record format on the READ or WRITE statement causes the physical read, write, or setup operations of a subfile to take place.

For a description of how the records in a subfile can be displayed and for a description of the keywords that can be specified for a subfile, see the *Programming: Data Description Specifications Reference*.

Some typical uses of subfiles include:

- Display only. The user reviews the display.
- Display with selection. The user requests more information about one of the items on the display.
- Modification. The user modifies one or more of the records.
- Input only, with no validity checking. A subfile is used for a data entry function, and the records are not checked.
- Input only, with validity checking. A subfile is used for a data entry function, but the records are checked.
- Combination of tasks. A subfile can be used as a display with modification.

Other Types of Device Files

Other types of device files include:

- DDM files
- Communications files
- BSC files
- Inline files.

DDM Files

Distributed Data Management (DDM) allows you to access data files that reside on personal computers or on remote IBM System/370, System/36, System/38, and AS/400 Systems. The PL/I compiler supports DDM files. You can retrieve, add, update or delete data records in a file that resides on another system. In addition, a remote system can access your AS/400 data base for record retrieval.

To use DDM files with PL/I programs, some important considerations are:

- When a DDM file is accessed as the source file for a PL/I program, the margins used in the compilation of the PL/I source are the default values of 2 and 72. No other margin values can be specified.

For more information about accessing remote files, refer to the *Communications: Distributed Data Management User's Guide*.

Communications and BSC Files

A communications file is a device file used to support Synchronous Data Link Control (SDLC) protocols. A BSC file is a device file used to support Binary Synchronous Communications, a form of communications line control that uses transmission control characters to control the transfer of data over a communication line.

Inline Files

An inline data file is a data file that is included as part of a batch job when the job is read by a reader or by the CL commands SBMDBJOB and SBMDKTJOB. For further information on these commands, see the *Programming: Control Language Reference*. An inline data file is delimited within the job by a //DATA command at the beginning of the file and by an end-of-data delimiter (a user-defined character string or the default of //) at the end of the file.

The record length for inline files is 80 bytes for a data file and 92 bytes for a source file.

An inline data file can be either named or unnamed. For an unnamed inline data file, either QINLINE is specified as the file name in the //DATA command or no name is specified. For a named inline data file, a file name is specified.

A named inline data file has the following characteristics:

- It has a unique name within a job; no other inline file can have the same name.
- It can be used more than once in a job.
- Each time it is opened, it is positioned to the first record.

Using Record Formats

A file record format must be defined to the system before you can use it in your programs. This can be done in two ways — externally described and program-described. Data base, display, communications, BSC, and printer files may be either externally described or program-described. Diskette, inline, and tape files can only be program-described.

Note that actual file processing within the PL/I program is the same if the file is externally described or program-described.

Externally Described Record Formats

An externally described file is described at the field level to OS/400 through DDS. The description includes information about the type of file, such as data base or a device, and a description of each field and its attributes.

By using the %INCLUDE directive, you can bring the external descriptions of a record's fields and attributes into your program and, therefore, you will not have to define them in your program. (See "Using the %INCLUDE Directive for External File Descriptions" on page 8-73.)

Externally described files offer the following advantages:

- Less coding in PL/I programs. If the same file is used by many programs, the fields can be defined once to OS/400 and used by all the programs. This eliminates the need to code record descriptions for PL/I programs that use externally described files.
- Less maintenance activity when the file's record format is changed. You can often update programs by changing the file's record format and then recompiling the programs that use the files without changing any coding in the program.
- Improved documentation because programs using the same files use consistent record format and field names.

Defining Record Formats with DDS

Externally described data files are described using DDS (Data Description Specifications). Using DDS, you provide descriptions of your files (including attributes of each field as well as record and file level information) used when the files are created. The Data Description Specifications form provides a common format for describing data externally.

There are two methods of providing information with the DDS form. You specify information that is frequently required by using the fixed columns provided on the form. You specify information that is less frequently needed by using the appropriate keywords and parameters. For example, the keyword DESCEND changes the sequencing of records from ascending (the normal sequence used by the system) to descending for a particular key field.

USING RECORD FORMATS

Level Checking

When a PL/I program uses an externally described file, the AS/400 System provides a level check function. This function ensures that at run time the format has not changed since compilation time.

If you specify the DESCRIBED option of the ENVIRONMENT attribute when you use the %INCLUDE directive to copy a record description of an externally described file into your program, level checking will occur for the copied record formats.

The level check function can be requested on the CL create, change, and override file commands. The default on the create file command is to request level checking. If level checking was requested, level checking occurs on a record format basis when the file is opened. If a level check error occurs, an UNDEFINEDFILE condition occurs at OPEN time.

If an existing format is used in a new file, any existing PL/I programs that use the format can still be used with the new file (assuming that no other conflicts such as a change of keys exist) without recompilation.

For more information on how to specify level checking, see the *Programming: Control Language Reference*.

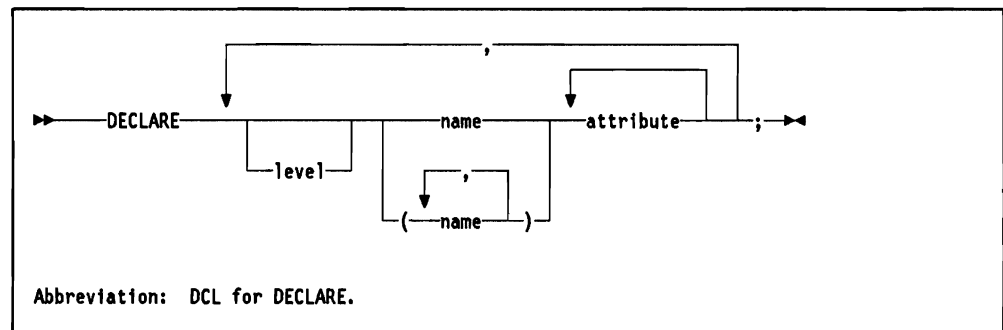
Program-Described Files

A program-described file is described within the PL/I program with DECLARE statements. The description of the file to OS/400 includes information about the type of file and the length of the records in the file.

Chapter 7. File Declaration and Input/Output

This chapter provides information about features of PL/I that are specific to AS/400 input/output. These features are discussed with reference to the ENVIRONMENT file declaration attribute, opening and closing files, and the OPTIONS option for input/output statements.

The following shows the syntax for the DECLARE statement, and how the ENVIRONMENT attribute fits into the overall file declaration. Refer to Chapter 12, "Declaring Names and Attributes of Variables" for more detailed information on the DECLARE statement.



The ENVIRONMENT Attribute

The ENVIRONMENT file declaration attribute allows you to specify information about implementation-defined features such as:

- File organization
- File characteristics
- File locking
- Commitment control
- Level Checking.

Options specified on the ENVIRONMENT attribute do not influence any default file attributes.

The following examples show several different reasons for using ENVIRONMENT options:

- An option may be required for successful data transmission.

For example, you must specify the INTERACTIVE option before you can open a display file.

- An option may be used to supply information.

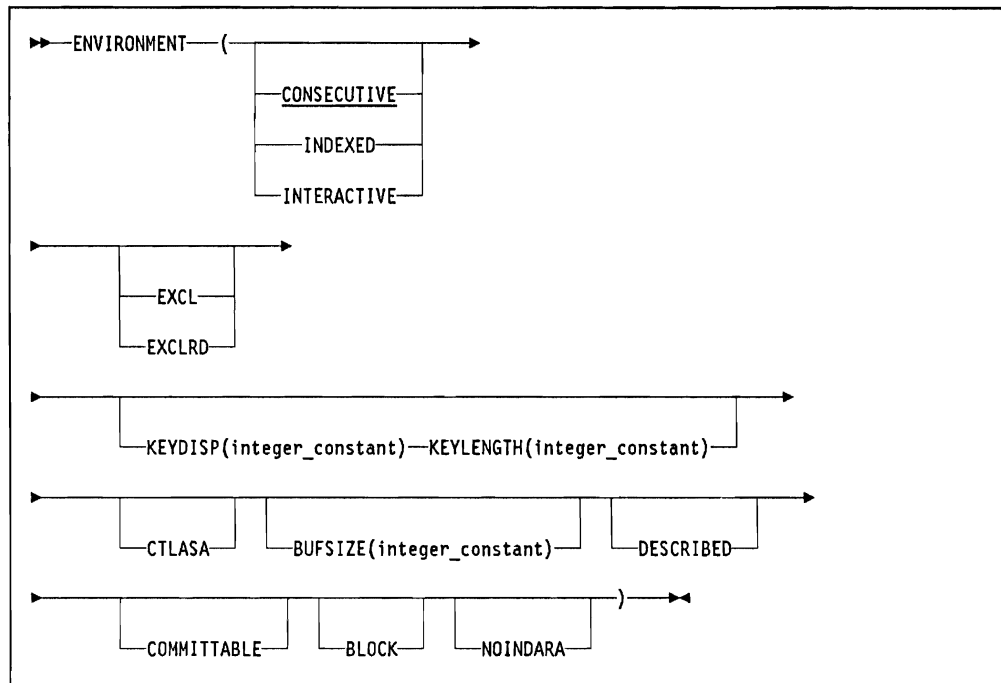
For example, you can use the BUFSIZE option to specify the record length of a file and override the system description of the file.

THE ENVIRONMENT ATTRIBUTE

- An option may be used to check appropriate association of program and system functions.

For example, when you specify the DESCRIBED option, the system processes level checking functions. See the *Programming: Data Management Guide* for a discussion of level checking.

The syntax of the ENVIRONMENT attribute is:



The options are discussed in the following sections.

File Organization Options

The file organization options are CONSECUTIVE, INDEXED, and INTERACTIVE. CONSECUTIVE is the default.

CONSECUTIVE

This indicates that records in the file are stored in a consecutive manner. A TAPE file for example, has CONSECUTIVE file organization. CONSECUTIVE is supported for all file types to increase device independence and allow for file redirection.

If you specify CONSECUTIVE for a logical file member that is based on more than one physical file member, the UNDEFINEDFILE condition is raised when the file is opened.

INDEXED

This indicates that the keyed access path for a physical or logical data base file is used.

You must explicitly code INDEXED to use a keyed sequence access path. If you do not code INDEXED, the default of CONSECUTIVE applies, the keyed

sequence access path is ignored, and the arrival sequence access path is used instead.

INTERACTIVE

This indicates that an interactive device is used. An interactive device is defined as either a display device or a remote program that is accessed through either a communications or BSC file. If you specify INTERACTIVE, you can use both READ and WRITE statements on a SEQUENTIAL UPDATE file.

File Locking Options

The file locking options, EXCL and EXCLRD, control the lock type applied to a data base file.

EXCL

Specifies that programs in concurrently running jobs cannot access the file.

EXCLRD

Specifies that programs in concurrently running jobs cannot update the file, but may read from the file.

These specifications apply to physical and logical data base files, not to display or device files.

When you specify a file locking option, the file member is locked at the time the file is opened. The lock is only visible to other concurrently running jobs. When you specify a file locking option for a logical file, one or more members of one or more physical files are locked.

If you do not specify one of these options, there will be a default lock on the file, depending on if the INPUT, OUTPUT, or UPDATE attribute is specified. If you specify the INPUT attribute, the default lock is shared read (*SHRRD); if you specify the OUTPUT or UPDATE attribute, the default lock is shared update (*SHRUPD). See "File Locking" on page 6-6, and the *Programming: Data Management Guide* for more information on file locking.

Key Options

The key options, KEYDISP and KEYLENGTH, move the KEYFROM variable on a WRITE statement or the KEY expression on a REWRITE statement into the record to ensure that the imbedded key and the associated key are of the same value when a WRITE or REWRITE statement is processed.

For INDEXED organization with program-described files, you must specify KEYDISP and KEYLENGTH. Also, if you specify INDEXED, but do not specify DESCRIBED, you must specify KEYDISP and KEYLENGTH.

KEYDISP and KEYLENGTH imply INDEXED.

KEYFROM(expression) can be used on a WRITE statement only if KEYDISP and KEYLENGTH are specified.

THE ENVIRONMENT ATTRIBUTE

If KEYDISP and KEYLENGTH are specified, the KEYFROM variable will be moved into the RECORD prior to the WRITE. Also, the KEY expression is moved into the record prior to the REWRITE.

KEYDISP (integer_constant)

Describes where the key field is located in the record. If you specify KEYDISP, do not specify DESCRIBED, CONSECUTIVE, or INTERACTIVE.

Use this option only for data base files in which all key fields are contiguous and each record format has the same key field definitions. KEYLENGTH must be specified when KEYDISP is specified.

The integer_constant is the displacement of the key in the record. For example, KEYDISP(0) is the first byte of the record. KEYDISP(1) is the second byte.

KEYLENGTH (integer_constant)

Describes the key length for the data base file. KEYLENGTH must be specified with KEYDISP. See KEYDISP above for details. KEYLENGTH must match the maximum key length for the data base file.

CTLASA Option

The CTLASA option specifies that the first byte of each record is expected to be a print control character (i.e. ANSI forms control character).

If you want to use this option, you must declare it explicitly. You are responsible for ensuring that the first byte of each record contains a valid print control character. The printer file that you direct the output to must have CTLCHAR(*FCFC) specified on the CL commands CRTPRTF or OVRPRTF in order to produce printer spacing corresponding to the print control characters in the printer file. No compiler support is provided other than activating First Character Forms Control (FCFC) support at file open time.

The valid print control characters are shown in the following table.

| Code | Action |
|---------|--------------------------------|
| (blank) | Space 1 line before printing |
| 0 | Space 2 lines before printing |
| - | Space 3 lines before printing |
| + | Suppress space before printing |
| 1 | Skip to channel 1 |
| 2 | Skip to channel 2 |
| 3 | Skip to channel 3 |
| 4 | Skip to channel 4 |
| 5 | Skip to channel 5 |

Figure 7-1 (Part 1 of 2). American National Standard Print Control Characters (CTLASA)

| Code | Action |
|------|--------------------|
| 6 | Skip to channel 6 |
| 7 | Skip to channel 7 |
| 8 | Skip to channel 8 |
| 9 | Skip to channel 9 |
| A | Skip to channel 10 |
| B | Skip to channel 11 |
| C | Skip to channel 12 |

Figure 7-1 (Part 2 of 2). American National Standard Print Control Characters (CTLASA)

You can also use the CL command OVRPRTF (Override Printer File) to set channel and line numbers (refer to the *Programming: Control Language Reference* for details).

Do not specify CTLASA for an externally described printer file. Use indicators instead of print control characters to provide spacing and other printer controls (see "Indicators" on page 8-76).

BUFSIZE (integer_constant) Option

The BUFSIZE option specifies a value that the program uses as the maximum record length for the file. The value specified is the input and output buffer length that is allocated for the file when it is opened.

You must code BUFSIZE when creating a new file on a diskette device: BUFSIZE specifies the record length on the diskette.

You can use the BUFSIZE option in the following ways:

- To override the actual file record length without changing the file specification.
- To specify the storage size in the system buffer.
- To specify the record length when creating diskette files.

If you do not specify BUFSIZE, the length of the input/output buffers is based on the maximum record length defined for the file.

The INTO option of the READ statement controls the amount of data read into the input buffer, up to a maximum you can specify on the BUFSIZE option.

The FROM option of a WRITE or REWRITE statement controls the amount of data written or rewritten from the output buffer, up to a maximum you can specify on the BUFSIZE option.

When a READ statement with the SET option is processed, any buffer storage beyond the BUFSIZE value is undefined.

THE ENVIRONMENT ATTRIBUTE

Note: If you specify the SET option of a READ statement, the program description of the record must match the actual records in the file, or unpredictable results may occur. There is a possibility of addressing beyond the end of the system input buffer, or if blocking is in effect, to address a portion of the next record in the buffer. To avoid this, specify the BUFSIZE option with a length equal to the longest record used for the file. This allows the redirection of the file to a file with a smaller record length at run time.

For tape files, BUFSIZE corresponds to the RCDLEN parameter on the CL command CRTTAPF (Create Tape File) or OVRTAPF (Override Tape File). Refer to the *Programming: Control Language Reference* for a discussion of the restrictions of the RCDLEN parameter with tape files.

DESCRIBED Option

The DESCRIBED option indicates that external record format definitions from the AS/400 file object are being used within the program.

The default is not to use DESCRIBED.

When you specify DESCRIBED, the following functions occur:

- Level checking is processed at time of file opening for all record formats associated with this file that have been declared using the %INCLUDE directive.
- Compile-time diagnostics are processed to ensure that the PL/I file attributes and system file attributes match. The valid combinations are shown in Figure 7-2 on page 7-9.

For files with indexed organization, you must ensure that the key is the same as the key in the replaced record. If DESCRIBED is specified and the values of the key fields in the record are not equal to the KEY expression, the results are unpredictable.

For DESCRIBED data base files, KEYFROM(*) must be coded on the WRITE statement when KEYFROM is used.

Note: You can prevent level checking by specifying LVLCHK(*NO) on one of the appropriate Create xxx File (CRTxxxF) or Override xxx File (OVRxxxF) commands. For more information on level checking, refer to the descriptions of the above commands in the *Programming: Control Language Reference*.

Commitment Control Option

The COMMITTABLE option indicates that a logical or physical data base file is placed under commitment control. Commitment control allows the user to ensure that multiple changes to data base files are all either made permanent or cancelled as a single operation when a commitment boundary is reached.

The default is not to use COMMITTABLE.

Use the COMMITTABLE option with the PLICOMMIT and PLIROLLBACK built-in subroutines (see "Using the PLICOMMIT Built-In Subroutine" on

page 8-59 and "Using the PLIROLLBACK Built-In Subroutine" on page 8-60) or their equivalents in other high level languages, or the CL commands STRCMTCTL (Start Commitment Control) and ENDCMTCTL (End Commitment Control). (See the *Programming: Control Language Reference*.)

The UNDEFINEDFILE condition is raised if COMMITTABLE is specified and the file is not a data base file.

Blocking Option

The BLOCK option specifies blocking for data base files.

If you specify BLOCK, you must not specify the OPTIONS option of the READ or WRITE statement.

BLOCK can be specified to improve performance when the file is primarily accessed by READ statements without the KEY option, or WRITE statements without the KEYFROM option.

When you specify BLOCK for an INPUT file, the complete block is transferred from the file to the input buffer. Records are read from the input buffer to the PL/I program, one at a time, until the buffer is empty.

When you specify BLOCK for an OUTPUT file, records are written from the program to the output buffer, one at a time, until the block is full. Then the complete block is transferred to the file.

The number of records in the block is optimized by the system and varies with each file type. Refer to the *Programming: Data Management Guide* for a description of the blocking support provided for each AS/400 file type.

If your application requires an immediate view of the file after each input or output transaction, do not specify blocking. If blocking is in effect for an INPUT file, new records are only available after a complete block is read into the program. If blocking is in effect for an OUTPUT file, new records are only available after a complete block is written to the data base. Therefore, duplicate key conditions may not be detected until blocks are transferred, or until files are closed. Programs sharing files will have to wait for newly added records until a block is written.

Even if you specify blocking, it is ignored under the following conditions:

- If you specify the file attributes DIRECT with INPUT or UPDATE.
- If you specify the ENVIRONMENT options INTERACTIVE or COMMITTABLE.
- If you specify an AS/400 file type that does not support blocking. Only data base, tape, and diskette files support blocking.

If you specify the BLOCK option for a data base file, and the option is ignored by OS/400, you will receive a diagnostic message when the file is opened. To find out when blocking is ignored for data base files, refer to the *Programming: Control Language Programmer's Guide*.

THE ENVIRONMENT ATTRIBUTE

NOINDARA Option

The NOINDARA option specifies that you are using indicators without the INDARA keyword specified in the DDS for the file.

If you specify NOINDARA, do not specify the INDICATORS option on record I/O statements.

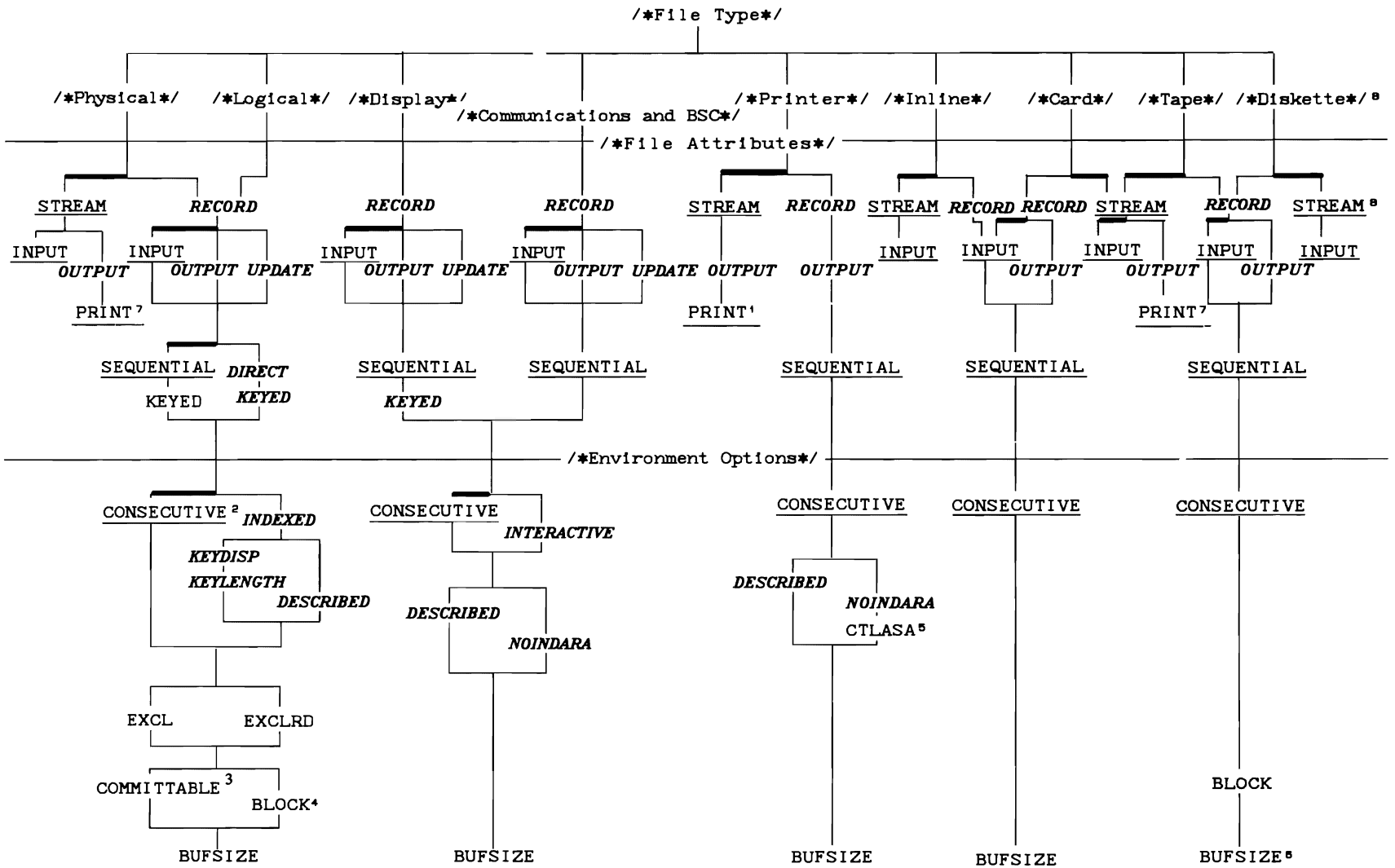
You must specify NOINDARA if you do not specify the DESCRIBED option of the ENVIRONMENT attribute and you do not specify INDARA in the DDS.

NOINDARA indicates that the external description of the file does not contain the DDS keyword INDARA. If NOINDARA is specified, both program-described files and externally described files are assumed to have indicators located in the record buffer, and not in a separate area. If NOINDARA is not specified, it will be assumed that the INDARA keyword has been specified in the DDS for the file. See "Using Indicators in the Record Area" on page 8-49.

If you specify DESCRIBED, PL/I will extract the file definition at compile time and use the information in the file to determine if INDARA is specified.

Figure 7-2 summarizes the valid combinations of file attributes and ENVIRONMENT options you can specify for the different types of AS/400 files. The figure notes explain how to use the figure.

Figure 7-2. Valid Combinations of File Attributes and Environment Options



THE ENVIRONMENT ATTRIBUTE

THE ENVIRONMENT ATTRIBUTE

Key:

| | |
|---------------|---|
| RECORD | must be declared explicitly or implicitly |
| <u>STREAM</u> | default attribute or option |
| <u> </u> | default path |
| /* */ | comment |

Footnotes:

- ¹STREAM and PRINT are not valid with an externally described print file.
- ²If you specify CONSECUTIVE for a logical file member that is based on more than one physical file member, the UNDEFINEDFILE condition is raised when the file is opened.
- ³File must have the EXTERNAL attribute.
- ⁴BLOCK is ignored if you specify DIRECT INPUT or UPDATE.
- ⁵CTLASA is not valid with an externally described print file.
- ⁶Required for a diskette file with RECORD OUTPUT.
- ⁷For physical and tape files, the forms control character is inserted in the first byte of each record by the PRINT attribute.
- ⁸Diskette files with STREAM OUTPUT are not allowed because diskette files do not have a default record length and it is not possible to specify a record length for diskette files (the ENVIRONMENT attribute can not be used with the STREAM attribute).

Additional Note:

For a list of implied attributes and options, see "Implied Attributes" on page 12-5.

How To Use The Table

Any line through the attributes and options in this table contains a complete set of the file description attributes for a name. By proceeding through the table, from top to bottom, you can select a valid combination of attributes and options for any line.

- Attributes and options shown in boldface must be declared explicitly (in a DECLARE statement) or implicitly (through the use of another attribute, option or statement). The first attribute or option in boldface on any line is required if you want any attributes or options on that line.
- Default attributes and options are underlined. They are selected for you, on the default path, unless you specify any alternative attributes or options (shown as branches from the default path).
- All other attributes and options are optional. You must specify them if you want the declared name to have that particular attribute or option.
- Exceptions to these points are discussed in the footnotes.

Opening and Closing Files

Before a file can be used for input or output, it must be opened. Opening a file involves locating the AS/400 file object through the library search list, the library name specified in the TITLE parameter of the OPEN statement or library specified on a corresponding override CL command. If the attempted connection is unsuccessful, the UNDEFINEDFILE condition is raised.

Scoping of Open Files (File Sharing)

Files within PL/I programs may have either an EXTERNAL or INTERNAL attribute. If you do not specify the scope, the default is EXTERNAL. When you declare a file with the EXTERNAL attribute, the file name is known to all PL/I procedures in the run unit. A file with the INTERNAL attribute can only be referenced in the block in which it is declared and in any blocks contained in the declaring block.

This section describes how the PL/I attribute of an EXTERNAL or INTERNAL file interacts with the system in determining how and when a file is opened. The sharing described here is only from the perspective of sharing from within one run unit. The section concludes with some special considerations for opening stream files and closing files after an error.

The first time a file is opened in a run unit, a new open data path (ODP) with a new invocation number is created for the file. Refer to the *Programming: Data Management Guide* for a description of the ODP.

Note: If you use the CL command RCLRSC (Reclaim Resource), you should be aware that this file belongs to the first PL/I procedure in the run unit, regardless of where it actually is located. Refer to the *Programming: Control Language Reference* for a description of the RCLRSC command.

Once a file is opened, additional OPEN statements for the file are ignored. The file retains the attributes with which it was first opened; no checking is processed to ensure that the attributes specified on the initial open are consistent with those specified on the OPEN statement that is ignored.

Once a file is opened, it remains open until one of the following occurs:

- You issue a CLOSE statement for the file.
- The run unit ends, and closes all the open files.

A file declared with the EXTERNAL attribute can be shared by programs in different run units if you specify SHARE(*YES) when you create the file. The file will remain open until the last program closes the file. If a file is closed following a run time error (see “Considerations for File Closing after an Error” on page 7-13), the actual closing of the file will not occur until all other concurrently running jobs using the file also issue close commands.

A file declared with the INTERNAL attribute cannot be shared, therefore you should specify SHARE(*NO) when you create the file. Even if you attempt to

OPENING AND CLOSING FILES

open an INTERNAL file with SHARE(*YES), the file is opened with SHARE(*NO).

Note: If you attempt to open an INTERNAL file that has specified SHARE(*YES), and that is already open within the job, the UNDEFINEDFILE condition is raised.

All INTERNAL PL/I files have their own separate connections to data management. Therefore, each INTERNAL file maintains its own file position, control information, and record buffers. All EXTERNAL files have one connection to data management. Therefore, all external files share the same control information, file position, and record buffers.

The set of records manipulated by either an INTERNAL or EXTERNAL file is established at the time the data management connection is made. This connection is made the first time the file is opened within the run unit. These rules are the same if the file is INTERNAL or EXTERNAL and only depend on what system file is connected to the INTERNAL or EXTERNAL PL/I file. The rules are:

- If the system file is a data base file or a named inline data file, all connections access the same set of records.
- If the system file is a spooled output file, each connection generates a unique set of spooled output.
- If the system file is a non-spooled device file, only one connection is allowed to the device at a time. In this case the records on the device are accessed directly.
- If the system file is the unnamed inline file QINLINE, each connection accesses the next set of unnamed inline data records.

Considerations for Opening a Print Stream File

When a print file is opened, the current position is at column 1 of line 1 of the first page. If your first PUT statement specifies that a certain number of lines are skipped, the skip will be based on this position. For instance, SKIP (3) indicates that printing begins on the fourth line, because it is the third line following the first line on the page.

Considerations for Opening a Non-Print Stream File

When a non-print stream file is opened, the current position is before the first record (the current position is at the end of an imaginary record immediately preceding the record accessed in the first GET or PUT statement). Therefore, if the first GET or PUT specifies, by means of a statement option or format item, that *n* lines are to be skipped before the first record is accessed, the current position will be at the start of record *n*.

Considerations for Opening SYSPRINT

When an action occurs that sends diagnostic output to SYSPRINT (for example, when you specify the GENOPT option *DIAGNOSE on the CL command CRTPLIPGM (Create PL/I Program)), this implicitly opens the file SYSPRINT with the attributes EXTERNAL, OUTPUT, STREAM, and PRINT. If you attempt to open SYSPRINT after this implicit opening, the attributes and options

you specify will be ignored, and any resulting incompatibilities will cause errors. This may be avoided if you declare SYSPRINT open with the INTERNAL scope attribute.

Considerations for File Closing after an Error

When a close operation involving certain device files (for example, BSC, Communications and Tape) occurs due to a run time error such as an operation with invalid data or a device error, PL/I must inform the system of the error condition so the system can correctly close the file. See the *Communications: Programmer's Guide* for details on closing files during error conditions.

PL/I will close a file with an error indication whenever a run unit ends and any of the following conditions are true:

- The job return code is greater than one.
- Control reaches the end of an ERROR on-unit.
- An ERROR condition is raised and no ERROR on unit exists.
- A STOP statement is processed.

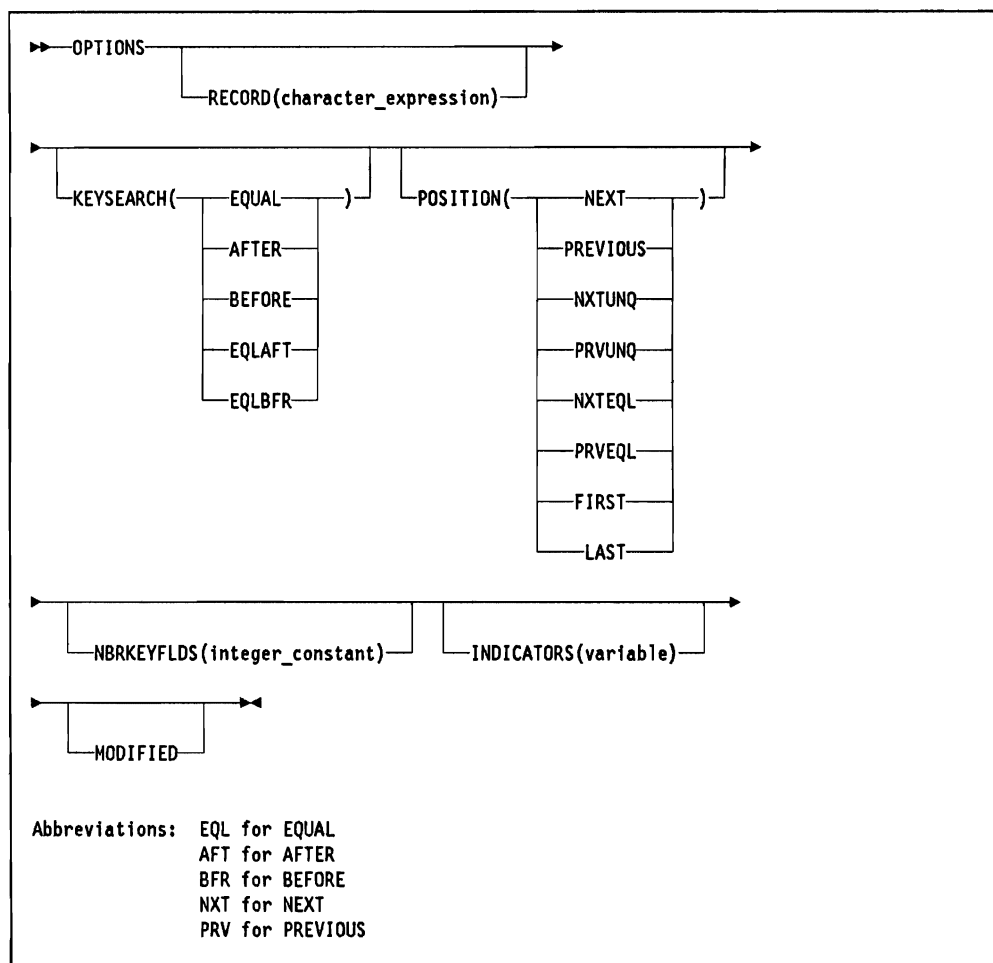
If you do not use a CLOSE statement to close a file, it is closed at the end of the run unit. For a complete description of implicit file closing refer to the section: "Scoping of Open Files (File Sharing)" on page 7-11.

If a CLOSE statement is processed within an ERROR on unit, the error close system interface is used. At compile time, if a called subroutine is within the scope of the ON unit block, and a called subroutine closes the file, an error close will occur. However, calls to subroutines outside the scope of the ON unit begin block will not close the file in error.

The OPTIONS Option of Record Data Transmission Statements

The OPTIONS option enhances the function of the record input/output statements READ, WRITE, REWRITE, and DELETE, to allow access to specific AS/400 data management functions. The syntax of the OPTIONS option is:

DATA TRANSMISSION OPTIONS



`OPTIONS` is not allowed if `BLOCK` is specified as an `ENVIRONMENT` option for the file.

The parameters of the `OPTIONS` option may be specified in any order. Figure 11-1 on page 11-17 shows valid combinations of option parameters and file organizations. Complete rules are shown in Appendix C, "Valid Combinations of Options for Input/Output Statements." The following sections describe the parameters of the `OPTIONS` option.

The `OPTIONS` parameters are discussed in the following order:

- `RECORD`
- `KEYSEARCH`
- `POSITION`
- `NBRKEYFLDS`
- `INDICATORS`
- `MODIFIED`.

RECORD Parameter

RECORD may be specified for a physical, logical, display, printer, BSC, or communications file that contains DDS record formats. It must be specified for a printer file with the ENVIRONMENT option DESCRIBED. It may be specified for any file organization for which the OPTIONS option is valid.

The RECORD parameter identifies a specific record format defined within a file object. The value of the character_expression is the name of the record format. You may use uppercase or lowercase letters, but the expression is converted to uppercase by the compiler. If the record format does not exist in the file to which the input/output statement is directed, the TRANSMIT condition is raised.

READ Using the RECORD Parameter

Every record in a file has a record format assigned to it. A file can have one or several record formats. When used with the READ statement, the RECORD parameter specifies the record format of the record which will be read in.

When used with a READ statement to a logical data base file that contains more than one record format, the RECORD option alters the search for the next record on the file access path. The next record read in will have the specified format, and any intervening records with different formats will be bypassed.

This is shown in Figure 7-3 for a SEQUENTIAL READ from a logical data base file with multiple record formats.

Logical File Records

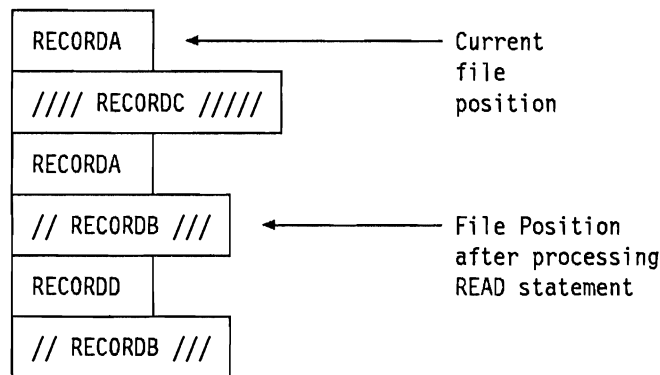


Figure 7-3. Example of RECORD Parameter

The logical file shown above contains four record formats, and the current file position is at RECORDA. If you process the statement:

```
READ ./././ OPTIONS (RECORD(RECORDB));
```

DATA TRANSMISSION OPTIONS

the record returned is the first record after the current file position that matches format B. If there are no records after the current file position that match format B, the ENDFILE condition is raised.

For physical and logical data base files, when the RECORD parameter is specified together with the KEY option or the KEYSEARCH, POSITION, or NBRKEYFLDS parameters, the following occurs. For the KEY option and the KEYSEARCH and NBRKEYFLDS parameters, the keyed access path is searched for a record that satisfies the key value optionally modified by the KEYSEARCH and NBRKEYFLDS rules. If a record is found, the RECORD value is compared for a matching record format name. If the record found does not match the RECORD value, additional access path entries are examined until a matching RECORD is found or a KEY condition is raised.

If you do not specify the RECORD parameter in a READ statement, a system-defined default is applied. The default values are given in the *Programming: Control Language Programmer's Guide*: they depend on the file type you are using. Also, the system will return the record format name of the last record read into the I/O feedback area. See "PLIIOFDB Built-In Subroutine" on page 15-16 for more details on how to access this information.

WRITE Using the RECORD Parameter

When used with the WRITE statement, the RECORD parameter specifies which format in the object file is used to create the new record in the file. You are responsible for ensuring that the data in the FROM option variable matches the record format description, because no check is made at run time.

RECORD is required for a WRITE to a multiple format logical file unless a format selection program is defined. (See the FMTSLR parameter on the CL command CRTLF (Create Logical File) in the *Programming: Control Language Reference*.)

When used with the WRITE statement to a display, printer, BSC, or communications file, the record format name supplied in the RECORD parameter determines the valid indicators supplied in the INDICATORS parameter. See "INDICATORS Parameter" on page 7-19.

REWRITE Using the RECORD Parameter

The use of RECORD with REWRITE is similar to its use with WRITE. You must specify RECORD if you are using a subfile. WRITE is optional with data base files, but it can provide useful documentation if you include it in your code. If you use a key, RECORD qualifies the search so that only records of the specified format are examined. If you do not use a key, the format specified must match that of the last record read. As with WRITE, you must ensure that the data being rewritten matches the record format specified, because no check is made at run time.

DELETE Using the RECORD Parameter

The use of RECORD with DELETE is the same as for REWRITE, except that RECORD cannot be specified when you are deleting a subfile record.

KEYSEARCH Parameter

The KEYSEARCH parameter of the READ statement applies only to files with INDEXED organization and SEQUENTIAL KEYED or DIRECT access. It enhances the key searching by allowing key comparisons other than key equal. The comparisons (except for EQUAL) are for key values either preceding or following the key value supplied. For example, if the keyed access path is descending, an option of AFTER will locate a key value that is lower in value than the key value supplied.

The KEYSEARCH parameter can be used with the NBRKEYFLDS and RECORD parameters. If the NBRKEYFLDS parameter is used, searching is restricted to the key values within the NBRKEYFLDS constant specified. If the RECORD parameter is used, the searching is restricted to the record format specified.

If KEYSEARCH is not specified, EQUAL is the default.

The KEYSEARCH values are:

| Value | Description |
|--------|---|
| EQUAL | Locate the first key (searching forwards) that is equal to the key value supplied in the KEY(expression). |
| AFTER | Locate the first key (searching forwards) that is after the key value supplied in the KEY(expression). |
| BEFORE | Locate the first key (searching backwards) that is before the key value supplied in the KEY(expression). |
| EQLAFT | Locate the first key (searching forwards) that is equal to the key value supplied in the KEY(expression). If no equal key exists, then locate the first key after the key value supplied in the KEY(expression). |
| EQLBFR | Locate the first key (searching forwards) that is equal to the key value supplied in the KEY(expression). If no equal key exists, then locate the first key (searching backwards) that is before the key value supplied in the KEY(expression). |

POSITION Parameter

The POSITION parameter of the READ statement cannot be used with INTERACTIVE organization or DIRECT access. Values NXTUNQ, PRVUNQ, NXTEQL, and PRVEQL can be used only with INDEXED organization. POSITION cannot be specified if KEY is specified. If RECORD is used with the POSITION parameter, only records with matching record formats are searched.

DATA TRANSMISSION OPTIONS

For information on using the POSITION parameter to read a file for which you have issued a CL override command, OVRDBF specifying MBR(*ALL), see "Members" on page 6-6.

The POSITION values are:

| Value | Description |
|----------|---|
| NEXT | Locate the next record in the access path. NEXT is the default if POSITION is not specified. |
| PREVIOUS | Read the previous record in the access path relative to the current file position. |
| NXTUNQ | Locate the next record in the access path that contains a different key value than the key value of the current file position. A current file position is required. If NBRKEYFLDS is specified, only key values within the number of fields specified are used to locate the next different key value. |
| PRVUNQ | Locate the nearest previous record in the access path that contains a different key value from the key value of the current file position. If NBRKEYFLDS is specified, only key values within the number of fields specified are used to locate the previous different key value. |
| NXTEQL | Locate the next record in the access path provided the key value of the next record is equal to the key value of the current file position. If the next record does not contain an equal key value, the KEY condition is raised. The test for an equal key value is made only within the NBRKEYFLDS specified or defaulted. |
| PRVEQL | Locate the previous record in the access path provided the key value of the previous record is equal to the key value of the current file position. If the previous record does not contain an equal key value, the KEY condition is raised. The test for an equal key value is made only within the NBRKEYFLDS specified or defaulted. |
| FIRST | Locate the first non-deleted record in the access path. |
| LAST | Locate the last non-deleted record in the access path. |

NBRKEYFLDS Parameter

The NBRKEYFLDS parameter of the READ statement may be used with INDEXED file organization. It specifies the number of key fields that are contained in the KEY expression.

If you do not specify the NBRKEYFLDS option, the length of the evaluated expression in the KEY expression is passed to the system. If the length is in between key fields, the system uses this length to process a generic key search. Refer to the *Programming: Control Language Programmer's Guide* for more information.

INDICATORS Parameter

The INDICATORS parameter of the READ, WRITE, and REWRITE statements may be used with INTERACTIVE organization, or with the READ or WRITE statement with CONSECUTIVE organization and SEQUENTIAL access. The file must contain external record definitions and must have the DDS INDARA keyword specified in order to use INDICATORS. Use of the DDS INDARA keyword is recommended for PL/I programs.

INDICATORS are used to communicate additional input/output information for display, printer, BSC, and communications files. The variable specified is used in a WRITE or REWRITE statement to set option indicators and to set response indicators after a READ statement. For more information on how to use indicators, refer to "Indicators" on page 8-76.

The variable specified with INDICATORS should contain one byte for each indicator defined for the record format. Each indicator number (from 1 to 99) corresponds to one byte in the variable. For example, indicator 1 is the first byte in the variable, indicator 5 is the fifth byte in the variable, and so on. The variable supplied with INDICATORS should be as long as the highest indicator defined for the record format. You can use the %INCLUDE directive to obtain a declaration of valid INDICATORS for each record format (see "Using the %INCLUDE Directive for External File Descriptions" on page 8-73).

On a WRITE or REWRITE statement using INDICATORS, the indicators defined for the record format must be set to either an on response of 'F1'X or an off response of 'F0'X. Indicators not defined for the record format are not examined. On a READ statement, the indicators defined for the record format returned to the program will be set to either an on-response of 'F1'X or an off-response of 'F0'X. Indicators not defined for the record format are not modified.

If the DDS keyword INDARA is not used in the external description of the file, but indicators are defined for the record, do not use the INDICATORS option to specify the presence of indicators. Furthermore, if the ENVIRONMENT option DESCRIBED is not specified, you must specify the ENVIRONMENT option NOINDARA.

If the DDS keyword INDARA is used in the external description of the file, and if the ENVIRONMENT option DESCRIBED is not specified, PL/I defaults to assuming the DDS keyword INDARA has been specified. For a WRITE or REWRITE, the indicators are in the output buffer. You must include the indicators in the record variable as part of the FROM option.

For a READ, the indicators are in the input buffer, and the INTO variable contains the indicators. If you use the SET option, the pointer-variable is set to the address of the first indicator in the input buffer. The indicators defined for the record format appear in the buffer in front of the data record. For examples showing the use of indicators, see "Example of Using Indicators" on page 8-43.

DATA TRANSMISSION OPTIONS

MODIFIED Parameter

The **MODIFIED** parameter of the **READ** statement may be used with **INTERACTIVE** organization and **SEQUENTIAL KEYED** access. This parameter applies to subfile processing. If **MODIFIED** is specified, the record read is the next subfile record that has been modified.

Chapter 8. Using AS/400 Files

After you have created the file and an external source file description (see the *Programming: Control Language Reference*, *Programming: Data Management Guide*, and the *Programming: Data Description Specifications Reference*), you must do the following in your program to use an AS/400 PL/I file:

1. Define the file in a DECLARE FILE statement.
2. Optionally include the record definitions with the %INCLUDE directive.
3. Open the file, either implicitly or explicitly.
4. Process the input and output.
5. Close the file, either implicitly or explicitly.

If you wish, you can then go through steps 3, 4, and 5 again.

Much of the information needed to carry out these operations is contained in Chapter 11, "Input and Output Statements," in Chapter 6, "AS/400 PL/I File and Record Management" and Chapter 7, "File Declaration and Input/Output." This chapter is concerned mainly with the relationship among all of this material.

This chapter is arranged in a series of examples illustrating the different types of data base files, and how they are updated, described, read, and so on. This chapter also contains information on commitment control and the %INCLUDE directive.

Using Data Base Files

A large amount of the input and output processed on the AS/400 System involves data base files. One of the unique aspects of data base files is discussed in Chapter 6, "AS/400 PL/I File and Record Management." The ENVIRONMENT options CONSECUTIVE and INDEXED, and the file attributes SEQUENTIAL, DIRECT, and KEYED, deal with access paths.

The ENVIRONMENT option CONSECUTIVE specifies that the file is to be processed using the arrival sequence access path. If you specify SEQUENTIAL access, you can only process the file sequentially: no relative record number keys can be specified. If you specify DIRECT access, you can obtain the record with the relative record number specified in the KEY or KEYFROM option of the input/output statement. If you specify SEQUENTIAL KEYED access, you can process the file either sequentially or by relative record numbers.

INDEXED specifies that the file is processed using the keyed sequence access path. If you specify SEQUENTIAL access, you can only process the file sequentially using the keyed sequence access path: you cannot specify a key. If you specify DIRECT access, you obtain a record with the key specified in the KEY or KEYFROM option of the input/output statement. If you specify SEQUENTIAL KEYED access, you can process the file either sequentially or by key.

USING DATA BASE FILES

With INDEXED organization, you also use the ENVIRONMENT attribute options KEYDISP and KEYLENGTH to process the key. The various input/output statement options you specify with each combination of file organization and access are shown in Appendix C, "Valid Combinations of Options for Input/Output Statements."

Also in the ENVIRONMENT option list, you have the option of specifying that you wish to use commitment control. You do this by declaring the COMMITTABLE option. Commitment control is described in "Commitment Control" on page 8-58.

Other ENVIRONMENT options which can be specified with data base files are BLOCK, BUFSIZE, EXCL, and EXCLRD. These options are described in Chapter 7, "File Declaration and Input/Output."

Externally Described Data Base Files

When using an externally described data base file with INDEXED organization, you have the option to specify the DESCRIBED option of the ENVIRONMENT attribute. This is discussed in Chapter 7, "File Declaration and Input/Output." Level checking is described in Chapter 6, "AS/400 PL/I File and Record Management."

For logical data base files, it is possible to map one record format to more than one base physical file member with the DTAMBRS parameter on the CL commands CRTLF and ADDLFM. A WRITE statement to a format of this type will fail.

Program-Described Data Base Files

In a program-described data base file, the record format or formats are described in the program. Even if there is a record format description in the DDS, it is ignored. When you describe a file in your program, you cannot use the %INCLUDE directive or the DESCRIBED attribute. There is no level checking when your program accesses the file, so you will not be protected against any changes in the record format made since you wrote your program. If you are using INDEXED files, you must specify KEYDISP and KEYLENGTH in your file declaration.

Data Description Specifications

For a full discussion of DDS coding and use, see the *Programming: Data Description Specifications Reference*. Examples are given here of a physical file DDS and a logical file DDS. The physical file, CUSMSTP, is used in the example in Figure 8-9 on page 8-23.

Example 1 - Describing a Physical File

| File | | Keying Instruction | | Graphic | | Description | | Page of | |
|------------|------|--------------------|-----|---------|-----|-------------|--|---------|--|
| Programmer | Date | Instruction | Key | Key | Key | | | | |

| Sequence Number | Form Type | Cond. Name | Name | Length | Reference (R) | Location | Functions |
|--|-----------|------------|--------|--------|---------------|----------|--|
| | | | | | | | |
| PHYSICAL CUSMSTP CUSTOMER MASTER FILE | | | | | | | |
| A | | R | CUSMST | | | | TEXT('CUSTOMER MASTER RECORD') |
| A | | | CUST | 5 | | | TEXT('CUSTOMER NUMBER') |
| A | | | NAME | 25 | | | TEXT('CUSTOMER NAME') |
| A | | | ADDR | 20 | | | TEXT('CUSTOMER ADDRESS') |
| A | | | CITY | 20 | | | TEXT('CUSTOMER CITY') |
| A | | | STATE | 2 | | | TEXT('STATE') |
| A | | | ZIP | 5 | 0 | | TEXT('ZIP CODE') |
| A | | | SRHCO | 6 | | | TEXT('CUSTOMER NUMBER SEARCH CODE') |
| A | | | CUSTYP | 1 | 0 | | TEXT('CUSTOMER TYPE 1-GOV 2-SCH + 3-BUS 4-PVT 5-OT') |
| A | | | ARBAL | 8 | 2 | | TEXT('ACCTS REC BALANCE') |
| A | | | ORDBAL | 8 | 2 | | TEXT('A/R AMT IN ORDER FILE') |
| A | | | LSTAMT | 8 | 2 | | TEXT('LAST AMT PAID IN A/R') |
| A | | | LSTDAT | 6 | 0 | | TEXT('LAST DATE PAID IN A/R') |
| A | | | CRDLMT | 8 | 2 | | TEXT('CUSTOMER CREDIT LIMIT') |
| A | | | SLSYR | 10 | 2 | | TEXT('CUSTOMER SALES THIS YEAR') |
| A | | | SLSLYR | 10 | 2 | | TEXT('CUSTOMER SALES LAST YEAR') |
| A | | K | CUST | | | | |

Figure 8-1. Physical Data Base File DDS

USING DATA BASE FILES

Example 2 – Describing a Logical File

| File | | Keying Instruction | | Graphic | | Description | | Page of | |
|------------|------|--------------------|-----|---------|-----|-------------|-----|---------|------|
| Programmer | Date | | | Key | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 |
| 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 |
| 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 |
| 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 |
| 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 100 |
| 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 |
| 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 |
| 121 | 122 | 123 | 124 | 125 | 126 | 127 | 128 | 129 | 130 |
| 131 | 132 | 133 | 134 | 135 | 136 | 137 | 138 | 139 | 140 |
| 141 | 142 | 143 | 144 | 145 | 146 | 147 | 148 | 149 | 150 |
| 151 | 152 | 153 | 154 | 155 | 156 | 157 | 158 | 159 | 160 |
| 161 | 162 | 163 | 164 | 165 | 166 | 167 | 168 | 169 | 170 |
| 171 | 172 | 173 | 174 | 175 | 176 | 177 | 178 | 179 | 180 |
| 181 | 182 | 183 | 184 | 185 | 186 | 187 | 188 | 189 | 190 |
| 191 | 192 | 193 | 194 | 195 | 196 | 197 | 198 | 199 | 200 |
| 201 | 202 | 203 | 204 | 205 | 206 | 207 | 208 | 209 | 210 |
| 211 | 212 | 213 | 214 | 215 | 216 | 217 | 218 | 219 | 220 |
| 221 | 222 | 223 | 224 | 225 | 226 | 227 | 228 | 229 | 230 |
| 231 | 232 | 233 | 234 | 235 | 236 | 237 | 238 | 239 | 240 |
| 241 | 242 | 243 | 244 | 245 | 246 | 247 | 248 | 249 | 250 |
| 251 | 252 | 253 | 254 | 255 | 256 | 257 | 258 | 259 | 260 |
| 261 | 262 | 263 | 264 | 265 | 266 | 267 | 268 | 269 | 270 |
| 271 | 272 | 273 | 274 | 275 | 276 | 277 | 278 | 279 | 280 |
| 281 | 282 | 283 | 284 | 285 | 286 | 287 | 288 | 289 | 290 |
| 291 | 292 | 293 | 294 | 295 | 296 | 297 | 298 | 299 | 300 |
| 301 | 302 | 303 | 304 | 305 | 306 | 307 | 308 | 309 | 310 |
| 311 | 312 | 313 | 314 | 315 | 316 | 317 | 318 | 319 | 320 |
| 321 | 322 | 323 | 324 | 325 | 326 | 327 | 328 | 329 | 330 |
| 331 | 332 | 333 | 334 | 335 | 336 | 337 | 338 | 339 | 340 |
| 341 | 342 | 343 | 344 | 345 | 346 | 347 | 348 | 349 | 350 |
| 351 | 352 | 353 | 354 | 355 | 356 | 357 | 358 | 359 | 360 |
| 361 | 362 | 363 | 364 | 365 | 366 | 367 | 368 | 369 | 370 |
| 371 | 372 | 373 | 374 | 375 | 376 | 377 | 378 | 379 | 380 |
| 381 | 382 | 383 | 384 | 385 | 386 | 387 | 388 | 389 | 390 |
| 391 | 392 | 393 | 394 | 395 | 396 | 397 | 398 | 399 | 400 |
| 401 | 402 | 403 | 404 | 405 | 406 | 407 | 408 | 409 | 410 |
| 411 | 412 | 413 | 414 | 415 | 416 | 417 | 418 | 419 | 420 |
| 421 | 422 | 423 | 424 | 425 | 426 | 427 | 428 | 429 | 430 |
| 431 | 432 | 433 | 434 | 435 | 436 | 437 | 438 | 439 | 440 |
| 441 | 442 | 443 | 444 | 445 | 446 | 447 | 448 | 449 | 450 |
| 451 | 452 | 453 | 454 | 455 | 456 | 457 | 458 | 459 | 460 |
| 461 | 462 | 463 | 464 | 465 | 466 | 467 | 468 | 469 | 470 |
| 471 | 472 | 473 | 474 | 475 | 476 | 477 | 478 | 479 | 480 |
| 481 | 482 | 483 | 484 | 485 | 486 | 487 | 488 | 489 | 490 |
| 491 | 492 | 493 | 494 | 495 | 496 | 497 | 498 | 499 | 500 |
| 501 | 502 | 503 | 504 | 505 | 506 | 507 | 508 | 509 | 510 |
| 511 | 512 | 513 | 514 | 515 | 516 | 517 | 518 | 519 | 520 |
| 521 | 522 | 523 | 524 | 525 | 526 | 527 | 528 | 529 | 530 |
| 531 | 532 | 533 | 534 | 535 | 536 | 537 | 538 | 539 | 540 |
| 541 | 542 | 543 | 544 | 545 | 546 | 547 | 548 | 549 | 550 |
| 551 | 552 | 553 | 554 | 555 | 556 | 557 | 558 | 559 | 560 |
| 561 | 562 | 563 | 564 | 565 | 566 | 567 | 568 | 569 | 570 |
| 571 | 572 | 573 | 574 | 575 | 576 | 577 | 578 | 579 | 580 |
| 581 | 582 | 583 | 584 | 585 | 586 | 587 | 588 | 589 | 590 |
| 591 | 592 | 593 | 594 | 595 | 596 | 597 | 598 | 599 | 600 |
| 601 | 602 | 603 | 604 | 605 | 606 | 607 | 608 | 609 | 610 |
| 611 | 612 | 613 | 614 | 615 | 616 | 617 | 618 | 619 | 620 |
| 621 | 622 | 623 | 624 | 625 | 626 | 627 | 628 | 629 | 630 |
| 631 | 632 | 633 | 634 | 635 | 636 | 637 | 638 | 639 | 640 |
| 641 | 642 | 643 | 644 | 645 | 646 | 647 | 648 | 649 | 650 |
| 651 | 652 | 653 | 654 | 655 | 656 | 657 | 658 | 659 | 660 |
| 661 | 662 | 663 | 664 | 665 | 666 | 667 | 668 | 669 | 670 |
| 671 | 672 | 673 | 674 | 675 | 676 | 677 | 678 | 679 | 680 |
| 681 | 682 | 683 | 684 | 685 | 686 | 687 | 688 | 689 | 690 |
| 691 | 692 | 693 | 694 | 695 | 696 | 697 | 698 | 699 | 700 |
| 701 | 702 | 703 | 704 | 705 | 706 | 707 | 708 | 709 | 710 |
| 711 | 712 | 713 | 714 | 715 | 716 | 717 | 718 | 719 | 720 |
| 721 | 722 | 723 | 724 | 725 | 726 | 727 | 728 | 729 | 730 |
| 731 | 732 | 733 | 734 | 735 | 736 | 737 | 738 | 739 | 740 |
| 741 | 742 | 743 | 744 | 745 | 746 | 747 | 748 | 749 | 750 |
| 751 | 752 | 753 | 754 | 755 | 756 | 757 | 758 | 759 | 760 |
| 761 | 762 | 763 | 764 | 765 | 766 | 767 | 768 | 769 | 770 |
| 771 | 772 | 773 | 774 | 775 | 776 | 777 | 778 | 779 | 780 |
| 781 | 782 | 783 | 784 | 785 | 786 | 787 | 788 | 789 | 790 |
| 791 | 792 | 793 | 794 | 795 | 796 | 797 | 798 | 799 | 800 |
| 801 | 802 | 803 | 804 | 805 | 806 | 807 | 808 | 809 | 810 |
| 811 | 812 | 813 | 814 | 815 | 816 | 817 | 818 | 819 | 820 |
| 821 | 822 | 823 | 824 | 825 | 826 | 827 | 828 | 829 | 830 |
| 831 | 832 | 833 | 834 | 835 | 836 | 837 | 838 | 839 | 840 |
| 841 | 842 | 843 | 844 | 845 | 846 | 847 | 848 | 849 | 850 |
| 851 | 852 | 853 | 854 | 855 | 856 | 857 | 858 | 859 | 860 |
| 861 | 862 | 863 | 864 | 865 | 866 | 867 | 868 | 869 | 870 |
| 871 | 872 | 873 | 874 | 875 | 876 | 877 | 878 | 879 | 880 |
| 881 | 882 | 883 | 884 | 885 | 886 | 887 | 888 | 889 | 890 |
| 891 | 892 | 893 | 894 | 895 | 896 | 897 | 898 | 899 | 900 |
| 901 | 902 | 903 | 904 | 905 | 906 | 907 | 908 | 909 | 910 |
| 911 | 912 | 913 | 914 | 915 | 916 | 917 | 918 | 919 | 920 |
| 921 | 922 | 923 | 924 | 925 | 926 | 927 | 928 | 929 | 930 |
| 931 | 932 | 933 | 934 | 935 | 936 | 937 | 938 | 939 | 940 |
| 941 | 942 | 943 | 944 | 945 | 946 | 947 | 948 | 949 | 950 |
| 951 | 952 | 953 | 954 | 955 | 956 | 957 | 958 | 959 | 960 |
| 961 | 962 | 963 | 964 | 965 | 966 | 967 | 968 | 969 | 970 |
| 971 | 972 | 973 | 974 | 975 | 976 | 977 | 978 | 979 | 980 |
| 981 | 982 | 983 | 984 | 985 | 986 | 987 | 988 | 989 | 990 |
| 991 | 992 | 993 | 994 | 995 | 996 | 997 | 998 | 999 | 1000 |

Figure 8-2. Logical Data Base File DDS

Example 3 - Writing Sequentially to a File with a Keyed Sequence Access Path

| Include | SEQNBR | STMT.SUBS | BLK | BN | DO | PL/I Source Listing | PLITST/LP1413 | 11/30/88 15:51:24 | Page 3 |
|---------|--------|-----------|-----|----|----|---|---------------|-------------------|-----------------|
| | | | | | | LP1413: PROCEDURE; | | | PUB00160 |
| | | | | | | *<.....1.....2.....3.....4.....5.....6.....7.....>.....8 Date | | | PUB00160 830817 |
| | | | | | | LP1413: PROCEDURE; | | | PUB00170 |
| | | | | | | /* FILE DECLARATIONS */ | | | PUB00180 |
| | | | | | | DECLARE | 1 | 2 | 3 |
| | | | | | | IN_FILE FILE RECORD INTERNAL SEQUENTIAL INPUT ENV(CONSECUTIVE | | | 4 |
| | | | | | | | | | 5 |
| | | | | | | | | | 6 |
| | | | | | | | | | 7 |
| | | | | | | | | | 8 |
| | | | | | | | | | 9 |
| | | | | | | | | | 10 |
| | | | | | | | | | 11 |
| | | | | | | | | | 12 |
| | | | | | | | | | 13 |
| | | | | | | | | | 14 |
| | | | | | | | | | 15 |
| | | | | | | | | | 16 |
| | | | | | | | | | 17 |
| | | | | | | | | | 18 |
| | | | | | | | | | 19 |
| | | | | | | | | | 20 |
| | | | | | | | | | 21 |
| | | | | | | | | | 22 |
| | | | | | | | | | 23 |
| | | | | | | | | | 24 |
| | | | | | | | | | 25 |
| | | | | | | | | | 26 |
| | | | | | | | | | 27 |
| | | | | | | | | | 28 |
| | | | | | | | | | 29 |
| | | | | | | | | | 30 |
| | | | | | | | | | 31 |
| | | | | | | | | | 32 |
| | | | | | | | | | 33 |
| | | | | | | | | | 34 |
| | | | | | | | | | 35 |
| | | | | | | | | | 36 |
| | | | | | | | | | 37 |
| | | | | | | | | | 38 |
| | | | | | | | | | 39 |
| | | | | | | | | | 40 |
| | | | | | | | | | 41 |
| | | | | | | | | | 42 |
| | | | | | | | | | 43 |
| | | | | | | | | | 44 |
| | | | | | | | | | 45 |
| | | | | | | | | | 46 |
| | | | | | | | | | 47 |
| | | | | | | | | | 48 |
| | | | | | | | | | 49 |
| | | | | | | | | | 50 |
| | | | | | | | | | 51 |
| | | | | | | | | | 52 |
| | | | | | | | | | 53 |
| | | | | | | | | | 54 |
| | | | | | | | | | 55 |
| | | | | | | | | | 56 |
| | | | | | | | | | 57 |
| | | | | | | | | | 58 |
| | | | | | | | | | 59 |
| | | | | | | | | | 60 |
| | | | | | | | | | 61 |
| | | | | | | | | | 62 |
| | | | | | | | | | 63 |
| | | | | | | | | | 64 |
| | | | | | | | | | 65 |
| | | | | | | | | | 66 |
| | | | | | | | | | 67 |
| | | | | | | | | | 68 |
| | | | | | | | | | 69 |
| | | | | | | | | | 70 |
| | | | | | | | | | 71 |
| | | | | | | | | | 72 |
| | | | | | | | | | 73 |
| | | | | | | | | | 74 |
| | | | | | | | | | 75 |
| | | | | | | | | | 76 |
| | | | | | | | | | 77 |
| | | | | | | | | | 78 |
| | | | | | | | | | 79 |
| | | | | | | | | | 80 |
| | | | | | | | | | 81 |
| | | | | | | | | | 82 |
| | | | | | | | | | 83 |
| | | | | | | | | | 84 |
| | | | | | | | | | 85 |
| | | | | | | | | | 86 |
| | | | | | | | | | 87 |
| | | | | | | | | | 88 |
| | | | | | | | | | 89 |
| | | | | | | | | | 90 |
| | | | | | | | | | 91 |
| | | | | | | | | | 92 |
| | | | | | | | | | 93 |
| | | | | | | | | | 94 |
| | | | | | | | | | 95 |
| | | | | | | | | | 96 |
| | | | | | | | | | 97 |
| | | | | | | | | | 98 |
| | | | | | | | | | 99 |
| | | | | | | | | | 100 |

Figure 8-3. Program Writing to a File with a Keyed Sequence Access Path

- 1 RECORD data transmission is used with IN_FILE.
- 2 The SEQUENTIAL access method is used with IN_FILE. Sequential reads and writes can be processed on any data base file if it has an arrival sequence access path or a keyed sequence access path.
- 3 IN_FILE is used for INPUT only: no output is directed to it.

USING DATA BASE FILES

- 4** CONSECUTIVE specifies that IN_FILE is processed using the arrival sequence access path.
- 5** BUFSIZE(38) indicates that the maximum record length is 38 characters. When the file is opened, this is the amount of storage the program allocates in the buffer for a record. If your data is blocked, your program will run faster if you specify BLOCK instead of BUFSIZE; this is explained in note 5 of Figure 8-4 on page 8-7.
- 6** RECORD data transmission is used with IND_FILE.
- 7** The SEQUENTIAL access method is used with IND_FILE.
- 8** IND_FILE is used for OUTPUT only: it provides no data to the program.
- 9** INDEXED specifies that IND_FILE has a keyed sequence access path.
- 10** KEYLENGTH(10) indicates that the key field is ten characters in length and KEYDISP(0) indicates that there are zero characters to the start of the key field from the beginning of the record. Therefore, the key is the first field in the record. KEYDISP and KEYLENGTH must be specified here, because the DESCRIBED option is not specified and the compiler is not drawing information on the key field from the external record format definition.
- 11** Because IND_FILE is being accessed sequentially, KEY is not specified with the WRITE statement, even though the file has a keyed sequence access path.

Example 4 - Updating a File with an Arrival Sequence Access Path

| 5728PL1 R01 M00 880715 | PL/I Source Listing | PLITST/LP1412 | 11/30/88 14:16:40 | Page 2 |
|------------------------|----------------------------|---|---|--|
| Include | SEQNBR STMT.SUBS BLK BN DO | LP1412: PROCEDURE; *<.....1.....2.....3.....4.....5.....6.....7.>.....8 Date LP1412: PROCEDURE; | | PUB00160 PUB00160 830817 PUB00170 PUB00180 PUB00190 PUB00200 830913 PUB00210 831114 PUB00210 831114 PUB00210 831114 830607 PUB00220 PUB00230 PUB00240 PUB00250 PUB00260 PUB00270 PUB00280 PUB00290 PUB00300 PUB00310 PUB00320 PUB00330 PUB00340 PUB00350 PUB00360 PUB00370 PUB00380 PUB00390 PUB00400 PUB00410 PUB00420 PUB00430 PUB00750 PUB00760 PUB00770 PUB00780 PUB00790 PUB00800 PUB00810 PUB00820 PUB00830 PUB00850 PUB00870 PUB00880 PUB00890 PUB00910 PUB00920 PUB00930 PUB00940 PUB00950 PUB00980 830919 PUB01000 PUB01010 |
| 100 | 1 | | | |
| 200 | | | | |
| 300 | | | /* FILE DECLARATIONS */ | |
| 400 | 2 | 1 1 | DECLARE | |
| 500 | | 1 1 | IN_FILE FILE RECORD INTERNAL SEQUENTIAL INPUT ENV(CONSECUTIVE | |
| 600 | | 1 1 | BLOCK), | |
| 700 | 2.1 | 1 1 | MST_FILE FILE RECORD INTERNAL SEQUENTIAL ENV(CONSECUTIVE | |
| 800 | | 1 1 | BLOCK), | |
| 900 | 2.2 | 1 1 | SYSPRINT FILE STREAM OUTPUT PRINT; | |
| 1000 | | | | |
| 1100 | | | /* RECORD DECLARATIONS */ | |
| 1200 | 3 | 1 1 | DECLARE | |
| 1300 | | 1 1 | 1 INPUT_EMPLOYEE, | |
| 1400 | 3.1 | 1 1 | 2 IN_EMP_NUMBER | PICTURE '999999', |
| 1500 | 3.2 | 1 1 | 2 IN_EMP_NAME | CHAR(28), |
| 1600 | 3.3 | 1 1 | 2 IN_EMP_CODE | PICTURE '9', |
| 1700 | 3.4 | 1 1 | 2 IN_EMP_SALARY | PICTURE '999999V99', |
| 1800 | 3.5 | 1 1 | 1 MASTER_EMPLOYEE, | |
| 1900 | 3.6 | 1 1 | 2 MST_EMP_NUMBER | PICTURE '999999', |
| 2000 | 3.7 | 1 1 | 2 MST_EMP_NAME | CHAR(28), |
| 2100 | 3.8 | 1 1 | 2 MST_EMP_CODE | PICTURE '9', |
| 2200 | 3.9 | 1 1 | 2 MST_EMP_SALARY | PICTURE '999999V99'; |
| 2300 | | | | |
| 2400 | | | /* PROGRAM FLAGS */ | |
| 2500 | 4 | 1 1 | DECLARE | |
| 2600 | | 1 1 | 1 BIT_FLAGS STATIC, | |
| 2700 | 4.1 | 1 1 | 2 MORE_RECORDS_INPUT | BIT(1) ALIGNED, |
| 2800 | 4.2 | 1 1 | 2 MORE_RECORDS_MASTER | BIT(1) ALIGNED, |
| 2900 | 4.3 | 1 1 | 2 NO | BIT(1) ALIGNED INIT('0'B), |
| 3000 | 4.4 | 1 1 | 2 YES | BIT(1) ALIGNED INIT('1'B); |
| 3100 | | | | |
| 3200 | 5 | 1 1 | ON ENDFILE (IN_FILE) | |
| 3300 | | 1 1 | MORE_RECORDS_INPUT = NO; | |
| 3400 | 6 | 1 1 | ON ENDFILE (MST_FILE) | |
| 3500 | | 1 1 | MORE_RECORDS_MASTER = NO; | |
| 3600 | | | | |
| 3700 | | | /* MAIN PROGRAM */ | |
| 3800 | 7 | 1 1 | MORE_RECORDS_INPUT = YES; | |
| 3900 | 8 | 1 1 | MORE_RECORDS_MASTER = YES; | |
| 4000 | | | | |
| 4100 | 9 | 1 1 | OPEN | |
| 4200 | | 1 1 | FILE (IN_FILE) TITLE('UPDATES'); /* INPUT */ | |
| 4300 | 10 | 1 1 | OPEN | |
| 4400 | | 1 1 | FILE (MST_FILE) UPDATE TITLE('MSTFILE'); | |
| 4500 | | | | |
| 4600 | 11 | 1 1 | READ FILE (IN_FILE) INTO (INPUT_EMPLOYEE); | |
| 4700 | 12 | 1 1 | READ FILE (MST_FILE) INTO (MASTER_EMPLOYEE); | |
| 4800 | | | | |
| 4900 | 13 | 1 1 | DO WHILE (MORE_RECORDS_INPUT & MORE_RECORDS_MASTER); | |
| 5000 | 14 | 1 1 1 | IF MST_EMP_NUMBER < IN_EMP_NUMBER THEN | |
| 5100 | | 1 1 1 | READ FILE (MST_FILE) INTO (MASTER_EMPLOYEE); | |
| 5200 | 15 | 1 1 1 | ELSE | |
| 5300 | | 1 1 1 | IF MST_EMP_NUMBER = IN_EMP_NUMBER THEN | |

Figure 8-4 (Part 1 of 2). Program Updating a File with an Arrival Sequence Access Path

USING DATA BASE FILES

| 5728PL1 R01 M00 880715 | PL/I Source Listing | PLITST/LP1412 | 11/30/88 14:16:40 | Page 3 |
|------------------------|----------------------------|--|--|-----------------|
| | LP1412: PROCEDURE; | | | PUB00160 |
| Include | SEQNBR STMT.SUBS BLK BN DO | *<..+...1...+...2...+...3...+...4...+...5...+...6...+...7>..+...8 Date | | |
| | 5400 | 1 1 1 | DO; | PUB01020 |
| | 5500 | 16 1 1 2 | 17 REWRITE FILE (MST_FILE) FROM (INPUT_EMPLOYEE); | PUB01040 |
| | 5600 | 17 1 1 2 | READ FILE (IN_FILE) INTO (INPUT_EMPLOYEE); | PUB01060 |
| | 5700 | 18 1 1 2 | READ FILE (MST_FILE) INTO (MASTER_EMPLOYEE); | PUB01070 |
| | 5800 | 19 1 1 2 | END; /* DO */ | PUB01080 |
| | 5900 | 20 1 1 1 | ELSE | PUB01090 |
| | 6000 | 1 1 1 | DO; | PUB01100 |
| | 6100 | 21 1 1 2 | 18 PUT FILE(SYSPRINT) SKIP EDIT ('ERROR--> INPUT RECORD ', | PUB01110 |
| | 6200 | 1 1 2 | IN_EMP_NUMBER, | PUB01120 |
| | 6300 | 1 1 2 | ' CANNOT BE FOUND IN THE MASTER FILE.') | PUB01130 |
| | 6400 | 1 1 2 | (A(23),X(2),F(6),X(2),A(36)); | PUB01140 |
| | 6500 | 22 1 1 2 | READ FILE (IN_FILE) INTO (INPUT_EMPLOYEE); | PUB01160 |
| | 6600 | 23 1 1 2 | END; /* DO */ | PUB01170 |
| | 6700 | 24 1 1 1 | END; /* DO WHILE */ | PUB01180 |
| | 6800 | | | PUB01190 |
| | 6900 | | /* APPEND ANY NEW RECORDS TO END OF MASTER FILE */ | PUB01200 |
| | 7000 | 25 1 1 | IF MORE_RECORDS_INPUT THEN | PUB01210 |
| | 7100 | 1 1 | DO; | PUB01220 |
| | 7200 | | /* CLOSE THE UPDATED MASTER FILE & REOPEN AS AN OUTPUT FILE */ | PUB01230 |
| | 7300 | 26 1 1 1 | CLOSE | PUB01250 |
| | 7400 | 1 1 1 | FILE (MST_FILE); | PUB01260 |
| | 7500 | 27 1 1 1 | OPEN | PUB01280 |
| | 7600 | 1 1 1 | FILE (MST_FILE) OUTPUT TITLE('MSTFILE'); | PUB01290 |
| | 7700 | | | PUB01300 |
| | 7800 | 28 1 1 1 | DO WHILE(MORE_RECORDS_INPUT); | PUB01310 |
| | 7900 | 29 1 1 2 | WRITE FILE (MST_FILE) FROM (INPUT_EMPLOYEE); | PUB01330 |
| | 8000 | 30 1 1 2 | READ FILE (IN_FILE) INTO (INPUT_EMPLOYEE); | PUB01350 |
| | 8100 | 31 1 1 2 | END; /* DO WHILE */ | PUB01360 |
| | 8200 | 32 1 1 1 | END; /* DO */ | PUB01370 |
| | 8300 | | | PUB01380 |
| | 8400 | 33 1 1 | CLOSE | PUB01400 |
| | 8500 | 1 1 | FILE (IN_FILE); | PUB01410 |
| | 8600 | 34 1 1 | CLOSE | PUB01420 |
| | 8700 | 1 1 | FILE (MST_FILE); | PUB01430 |
| | 8800 | | | PUB01440 |
| | 8900 | 35 1 1 | END LP1412; | PUB01450 830817 |

Figure 8-4 (Part 2 of 2). Program Updating a File with an Arrival Sequence Access Path

- 1 RECORD data transmission is used with IN_FILE.
- 2 The SEQUENTIAL access method is used with IN_FILE.
- 3 IN_FILE is used for INPUT only: no output is directed to it.
- 4 CONSECUTIVE specifies that IN_FILE is processed using the arrival sequence access path.
- 5 BLOCK specifies that instead of a single record, an entire block of data is read into or out of the buffer. Individual records are moved between the buffer and the program when required. This reduces the number of input/output operations needed, and therefore the program will run faster.
- 6 RECORD data transmission is used with MST_FILE.
- 7 The SEQUENTIAL access method is used with MST_FILE.
- 8 CONSECUTIVE specifies that MST_FILE is processed using the arrival sequence access path.
- 9 BLOCK specifies that each input/output operation moves a block of data to or from the buffer instead of a single record.

- 10** STREAM data transmission is used with SYSPRINT. Stream files can only be accessed sequentially.
- 11** SYSPRINT receives OUTPUT only. It provides no data to the program.
- 12** The PRINT attribute indicates that the first character in the record is an ASA printer control character.
- 13** IN_FILE is opened with the TITLE option specifying 'UPDATES'. Because the library and member names are allowed to default, the first member of file UPDATES in the library list is opened.
- 14** MST_FILE is opened with the TITLE option specifying 'MSTFILE'. Because the library and member names are allowed to default, the first member of file MSTFILE in the library list is opened.
- 15** The UPDATE attribute is specified for MST_FILE. There is no similar option specified for IN_FILE, because INPUT has already been specified as its data transmission mode in the file declaration. You can specify the data transmission mode for a file either in the file declaration or in the OPEN statement. If you specify the data transmission mode in both places, you must be sure that the mode specified is the same in both places. If you specify the mode in the OPEN statement only, you can close the file and then reopen it specifying a different data transmission mode.
- 16** You must process a READ on a non-keyed sequential UPDATE file before processing a REWRITE.
- 17** The REWRITE updates the record that was read at **16**.
- 18** SYSPRINT is implicitly opened by the PUT statement, because no OPEN statement is coded for it. Any file not already opened explicitly is implicitly opened by the first data transmission statement which accesses it. It is a good practice, however, to explicitly open all your files.

USING DATA BASE FILES

Example 5—Updating a File with a Keyed Access Path

```

5728PL1 R01 M00 880715          PL/I Source Listing          PLITST/LP1414          11/30/88 14:18:11          Page 2
LP1414: PROCEDURE;
Include  SEQNBR STMT.SUBS BLK BN DO  *<.....1.....2.....3.....4.....5.....6.....7.....8 Date
LP1414: PROCEDURE;
100      1
200
300      /* FILE DECLARATIONS */
400      2      1 1      DECLARE          1          2          3          4          PUB00190
500      1 1      IN_FILE FILE RECORD INTERNAL SEQUENTIAL INPUT ENV(CONSECUTIVE          5          PUB00200 830930
600      1 1      BUFSIZE(38)),          6          831114
700      2.1      1 1      MST_FILE FILE RECORD INTERNAL DIRECT UPDATE ENV(INDEXED          7          PUB00210 830919
800      1 1      KEYDISP(0) KEYLENGTH(10)),          10          831114
900      2.2      1 1      SYSPRINT FILE STREAM OUTPUT PRINT;          11          830607
1000
1100     /* RECORD DECLARATIONS */
1200     3      1 1      DECLARE          PUB00220
1300     1 1      1 MASTER_RECORD STATIC,          PUB00230
1400     3.1      1 1      2 MASTER_KEY,          PUB00240
1500     3.2      1 1      3 MASTER_GEN_FLD          CHAR(5),          PUB00250
1600     3.3      1 1      3 MASTER_DET_FLD          CHAR(5),          PUB00270
1700     3.4      1 1      2 MASTER_NAME          CHAR(20),          PUB00280
1800     3.5      1 1      2 MASTER_BAL          PICTURE '999999V9',          PUB00290
1900     3.6      1 1      1 INPUT_RECORD,          PUB00300
2000     3.7      1 1      2 INPUT_KEY,          PUB00310 831003
2100     3.8      1 1      3 INPUT_GEN_FLD          CHAR(5),          PUB00320
2200     3.9      1 1      3 INPUT_DET_FLD          CHAR(5),          PUB00330
2300     3.10     1 1      2 INPUT_NAME          CHAR(20),          PUB00340
2400     3.11     1 1      2 INPUT_AMT          PICTURE 'S99999V99';          PUB00350
2500
2600     /* PROGRAM FLAGS */
2700     4      1 1      DECLARE          PUB00360
2800     1 1      1 BIT_FLAGS STATIC,          PUB00370
2900     4.1      1 1      2 MORE_RECORDS          BIT(1) ALIGNED,          PUB00380
3000     4.2      1 1      2 NO          BIT(1) ALIGNED INIT('0'B),          PUB00390
3100     4.3      1 1      2 YES          BIT(1) ALIGNED INIT('1'B);          PUB00400
3200
3300     /* PROGRAM VARIABLES */
3400     5      1 1      DECLARE          PUB00410
3500     1 1      OLD_MASTER_BAL          PICTURE 'S99999V99',          PUB00420
3600     5.1      1 1      PAGE_NUMBER          BINARY FIXED(2);          PUB00430
3700
3800     ON ENDFILE(IN_FILE)          PUB00440
3900     MORE_RECORDS = NO;          PUB00450
4000
4100     ON ENDPAGE (SYSPRINT)          PUB00460
4200     BEGIN;          PUB00470
4300     8      3 2      14 PUT FILE (SYSPRINT) PAGE EDIT('PAGE ',PAGE_NUMBER)          PUB00480
4400     (X(81),A(6),F(2));          PUB00490
4500     9      3 2      PUT FILE (SYSPRINT) SKIP(3) EDIT('UPDATE REPORT')(X(38),A(13));          PUB00500
4600     10     3 2      PUT FILE (SYSPRINT) SKIP(2) EDIT('KEY ID','NAME','CUR BALANCE',          PUB00510
4700     'UPDATE AMOUNT','NEW BALANCE')(A(6),X(9),A(4),X(21),A(11),          PUB00520
4800     X(6),A(13),X(4),A(11));          PUB00530
4900     11     3 2      PAGE_NUMBER = PAGE_NUMBER + 1;          PUB00540
5000     12     3 2      END; /* BEGIN */          PUB00550
5100
5200     /* MAIN PROGRAM */          PUB00560
5300     13     1 1      PAGE_NUMBER = 1;          PUB00570

```

Figure 8-5 (Part 1 of 2). Program Updating a File with a Keyed Sequence Access Path

| 5728PL1 R01 M00 880715 | PL/I Source Listing | PLITST/LP1414 | 11/30/88 14:18:11 | Page 3 |
|------------------------|----------------------------|--|-------------------|-----------------|
| | LP1414: PROCEDURE; | | | PUB00160 |
| Include | SEQNBR STMT.SUBS BLK BN DO | *<.....1.....2.....3.....4.....5.....6.....7.....>.....8 Date | | |
| | 5400 14 1 1 | MORE_RECORDS = YES; | | PUB00670 |
| | 5500 | | | PUB00680 |
| | 5600 15 1 1 | OPEN 15 | | PUB00690 |
| | 5700 1 1 | FILE (IN_FILE) TITLE('UPDATES'); /* INPUT */ | | PUB00700 830929 |
| | 5800 16 1 1 | OPEN 16 | | PUB00710 |
| | 5900 1 1 | FILE (MST_FILE) TITLE('MSTFILE'); /* UPDATE */ | | PUB00720 830919 |
| | 6000 | | | PUB00730 |
| | 6100 17 1 1 | SIGNAL ENDPAGE (SYSPRINT); | | PUB00740 |
| | 6200 18 1 1 | READ FILE (IN_FILE) INTO (INPUT_RECORD); | | PUB00750 |
| | 6300 | | | PUB00760 |
| | 6400 19 1 1 | DO WHILE(MORE_RECORDS); | | PUB00770 |
| | 6500 20 1 1 1 | IF INPUT_DET_FLD = ' ' THEN | | PUB00780 |
| | 6600 1 1 1 | CALL INITSEQ; | | PUB00790 |
| | 6700 21 1 1 1 | ELSE | | PUB00800 |
| | 6800 1 1 1 | CALL DYNAMIC; | | PUB00810 |
| | 6900 22 1 1 1 | READ FILE (IN_FILE) INTO (INPUT_RECORD); | | PUB00820 |
| | 7000 23 1 1 1 | END; /* DO WHILE */ | | PUB00830 |
| | 7100 | | | PUB00840 |
| | 7200 24 1 1 | CLOSE | | PUB00850 |
| | 7300 1 1 | FILE (IN_FILE); | | PUB00860 |
| | 7400 25 1 1 | CLOSE | | PUB00870 |
| | 7500 1 1 | FILE (MST_FILE); | | PUB00880 |
| | 7600 | | | PUB00890 |
| | 7700 26 1 1 | INITSEQ: PROCEDURE; | | PUB00900 |
| | 7800 27 4 2 | MASTER_GEN_FLD = INPUT_GEN_FLD; | | PUB00910 |
| | 7900 28 4 2 | 17 READ FILE (MST_FILE) INTO (MASTER_RECORD) KEY(MASTER_KEY) | | PUB00920 830930 |
| | 8000 4 2 | OPTIONS(KEYSEARCH(EQLAFT) NBRKEYFLDS(1)); | | PUB00930 830930 |
| | 8100 29 4 2 | DO WHILE(INPUT_GEN_FLD = MASTER_GEN_FLD); | | PUB00940 |
| | 8200 30 4 2 1 | CALL SEQPROC; | | PUB00950 |
| | 8300 31 4 2 1 | END; /* DO WHILE */ | | PUB00960 |
| | 8400 32 4 2 | RETURN; | | PUB00970 |
| | 8500 33 4 2 | END INITSEQ; | | PUB00980 |
| | 8600 | | | PUB00990 |
| | 8700 34 1 1 | SEQPROC: PROCEDURE; | | PUB01000 |
| | 8800 35 5 2 | PUT FILE (SYSPRINT) SKIP EDIT(MASTER_KEY,MASTER_NAME, | | PUB01040 831003 |
| | 8900 5 2 | MASTER_BAL)(A(5),A(5),X(5),A(20),X(6),F(10,2)); | | PUB01050 831003 |
| | 9000 36 5 2 | 18 READ FILE (MST_FILE) INTO (MASTER_RECORD) KEY(MASTER_KEY) | | PUB01010 831003 |
| | 9100 5 2 | OPTIONS(KEYSEARCH(AFTER)); | | PUB01020 831003 |
| | 9200 37 5 2 | RETURN; | | PUB01060 |
| | 9300 38 5 2 | END SEQPROC; | | PUB01070 |
| | 9400 | | | PUB01080 |
| | 9500 39 1 1 | DYNAMIC: PROCEDURE; | | PUB01090 |
| | 9600 40 6 2 | MASTER_KEY = INPUT_KEY; | | PUB01100 |
| | 9700 41 6 2 | READ FILE (MST_FILE) INTO (MASTER_RECORD) KEY(MASTER_KEY) | | PUB01110 831003 |
| | 9800 6 2 | OPTIONS(KEYSEARCH(EQUAL)); | | 831003 |
| | 9900 42 6 2 | IF INPUT_GEN_FLD = MASTER_GEN_FLD THEN | | PUB01120 |
| | 10000 6 2 | DO; | | PUB01130 |
| | 10100 43 6 2 1 | OLD_MASTER_BAL = MASTER_BAL; | | PUB01140 |
| | 10200 44 6 2 1 | MASTER_BAL = MASTER_BAL + INPUT_AMT; | | PUB01150 |
| | 10300 45 6 2 1 | PUT FILE (SYSPRINT) SKIP EDIT(MASTER_KEY,MASTER_NAME, | | PUB01160 |
| | 10400 6 2 1 | OLD_MASTER_BAL,INPUT_AMT,MASTER_BAL)(A(5),A(5),X(5),A(20), | | PUB01170 831003 |
| | 10500 6 2 1 | X(6),F(10,2),X(6),F(10,2),X(8),F(10,2)); | | PUB01180 831003 |
| | 10600 46 6 2 1 | 19 REWRITE FILE (MST_FILE) FROM (MASTER_RECORD) KEY(MASTER_KEY); | | PUB01190 |
| | 10700 47 6 2 1 | END; /* DO */ | | PUB01200 |
| | 10800 48 6 2 | RETURN; | | PUB01210 |
| | 10900 49 6 2 | END DYNAMIC; | | PUB01220 |
| | 11000 | | | PUB01230 |
| | 11100 50 1 1 | END LP1414; | | PUB01240 830817 |

Figure 8-5 (Part 2 of 2). Program Updating a File with a Keyed Sequence Access Path

- 1 RECORD data transmission is used with IN_FILE.
- 2 The SEQUENTIAL access method is used with IN_FILE.
- 3 IN_FILE is used for INPUT only: no output is directed to it.

USING DATA BASE FILES

- 4** CONSECUTIVE specifies that IN_FILE is processed using the arrival sequence access path.
- 5** BUFSIZE(38) indicates that the maximum record length is 38 characters. When the file is opened this is the amount of storage the program allocates in the buffer for a record. If your data is blocked, the program will run faster if you specify BLOCK instead of BUFSIZE; this is explained in note 5 of Figure 8-4 on page 8-7.
- 6** RECORD data transmission is used with MST_FILE.
- 7** The DIRECT access method is used with MST_FILE. MST_FILE is accessed non-sequentially. The target record is located by a key.
- 8** MST_FILE has the UPDATE attribute, because the program will receive data from it and also direct data to it.
- 9** INDEXED specifies that MST_FILE is processed using the keyed sequence access path.
- 10** KEYLENGTH(10) indicates that the key field is ten characters in length and KEYDISP(0) indicates that there are no characters between the start of the key field and the beginning of the record. Therefore, the key is the first field in the record. KEYDISP and KEYLENGTH must be specified here, because the DESCRIBED option is not specified and the compiler is not drawing information on the key field from the external record format definition.
- 11** STREAM data transmission is used with SYSPRINT. SYSPRINT is accessed sequentially, but this is not coded as an attribute because stream data transmission can only use sequential access.
- 12** SYSPRINT is used for OUTPUT only. It provides no input to the program.
- 13** The PRINT attribute specifies that the first character in every record is an ASA printer control character.
- 14** SYSPRINT is not explicitly opened by an OPEN statement; it is therefore implicitly opened the first time the PUT statement is processed. Any file which has not been explicitly opened is implicitly opened by the first data transmission statement which accesses it. It is a good practice, however, to explicitly open all your files.

SYSPRINT is the default file in STREAM OUTPUT files. The PUT statement will therefore compile and process correctly if FILE (SYSPRINT) is omitted.
- 15** IN_FILE is opened with the TITLE option specifying 'UPDATES'; because the library and member names are allowed to default, the first member of file UPDATES in the library list is opened.
- 16** MST_FILE is opened with the TITLE option specifying 'MSTFILE'; because the library and member names are allowed to default, the first member of file MSTFILE in the library list is opened.

- 17** The READ statement uses MASTER_KEY to find a record in MST_FILE and read it into MASTER_RECORD. The KEYSEARCH option specifies EQLAFT. If a record with a precisely equal key is not found, the record with the next highest key is read into MASTER_RECORD. The NBRKEYFLDS option is specified and given a parameter value of 1, indicating that only one key is used in the search for the record.
- 18** The next record is read in from MASTER_FILE, using MASTER_KEY. KEYSEARCH(AFTER) specifies that the record with the next highest key after the current record is read in.
- 19** The record read in from MST_FILE is rewritten, using MASTER_KEY.

Example 6 - Writing to an Arrival Sequence File by Relative Record Number

| 5728PL1 R01 M00 880715 | PL/I Source Listing | PLITST/LP1415 | 11/30/87 14:18:54 | Page 2 |
|------------------------|----------------------------|---|----------------------------------|--|
| Include | SEQNBR STMT.SUBS BLK BN DO | LP1415: PROCEDURE; *<.....1.....2.....3.....4.....5.....6.....7.>.....8 Date LP1415: PROCEDURE; | | PUB00160 PUB00160 830817 PUB00170 PUB00180 PUB00190 PUB00200 831004 PUB00210 PUB00220 PUB00230 PUB00240 PUB00250 PUB00260 PUB00270 831114 PUB00280 831114 PUB00290 PUB00300 PUB00310 PUB00320 831114 PUB00330 831114 PUB00340 PUB00350 PUB00360 PUB00370 PUB00380 PUB00390 PUB00400 PUB00410 PUB00420 PUB00430 PUB00440 PUB00450 PUB00460 PUB00470 PUB00480 PUB00490 PUB00500 PUB00510 PUB00520 PUB00530 PUB00540 PUB00550 PUB00560 830919 PUB00570 PUB00580 831003 PUB00590 PUB00600 PUB00610 PUB00620 PUB00630 PUB00640 PUB00650 PUB00660 PUB00670 |
| 100 | 1 | | | |
| 200 | | | | |
| 300 | | /* FILE DECLARATIONS */ | | |
| 400 | 2 | 1 1 DECLARE | 1 2 3 4 | |
| 500 | | 1 1 IN_FILE FILE RECORD INTERNAL | SEQUENTIAL INPUT ENV(CONSECUTIVE | |
| 600 | | 1 1 | BUFSIZE(23)),5 | |
| 700 | 2.1 | 1 1 REL_FILE FILE RECORD INTERNAL | DIRECT OUTPUT ENV(CONSECUTIVE); | |
| 800 | | | 6 7 8 9 | |
| 900 | | /* RECORD DECLARATIONS */ | | |
| 1000 | 3 | 1 1 DECLARE | | |
| 1100 | | 1 1 1 RELATIVE_RECORD_01, | | |
| 1200 | 3.1 | 1 1 2 RELATIVE_RECORD(5), | | |
| 1300 | 3.2 | 1 1 3 REL_YEAR | PICTURE '99', | |
| 1400 | 3.3 | 1 1 3 REL_WEEK | PICTURE '99', | |
| 1500 | 3.4 | 1 1 3 REL_UNIT_SALES | PICTURE 'S999999', | |
| 1600 | 3.5 | 1 1 3 REL_DOLLAR_SALES | PICTURE 'S999999999V99', | |
| 1700 | 3.6 | 1 1 1 INPUT_RECORD, | | |
| 1800 | 3.7 | 1 1 2 INPUT_YEAR | PICTURE '99', | |
| 1900 | 3.8 | 1 1 2 INPUT_WEEK | PICTURE '99', | |
| 2000 | 3.9 | 1 1 2 INPUT_UNIT_SALES | PICTURE 'S999999', | |
| 2100 | 3.10 | 1 1 2 INPUT_DOLLAR_SALES | PICTURE 'S999999999V99'; | |
| 2200 | | | | |
| 2300 | | /* PROGRAM FLAGS */ | | |
| 2400 | 4 | 1 1 DECLARE | | |
| 2500 | | 1 1 BIT_FLAGS STATIC, | | |
| 2600 | 4.1 | 1 1 2 MORE_RECORDS | BIT(1) ALIGNED, | |
| 2700 | 4.2 | 1 1 2 NO | BIT(1) ALIGNED INIT('0'B), | |
| 2800 | 4.3 | 1 1 2 YES | BIT(1) ALIGNED INIT('1'B); | |
| 2900 | | | | |
| 3000 | | /* PROGRAM VARIABLES */ | | |
| 3100 | 5 | 1 1 DECLARE | | |
| 3200 | | 1 1 REL_INDEX | BINARY FIXED(2); | |
| 3300 | | | | |
| 3400 | 6 | 1 1 ON ENDFILE(IN_FILE) | | |
| 3500 | | 1 1 MORE_RECORDS = NO; | | |
| 3600 | | | | |
| 3700 | | /* MAIN PROGRAM */ | | |
| 3800 | 7 | 1 1 MORE_RECORDS = YES; | | |
| 3900 | 8 | 1 1 REL_INDEX = 1; | | |
| 4000 | | | | |
| 4100 | 9 | 1 1 OPEN 10 | | |
| 4200 | | 1 1 FILE (IN_FILE) TITLE('INFILE'); /* INPUT */ | | |
| 4300 | 10 | 1 1 OPEN 11 | | |
| 4400 | | 1 1 FILE (REL_FILE) TITLE('MSTFILE'); /* OUTPUT */ | | |
| 4500 | | | | |
| 4600 | 11 | 1 1 READ FILE (IN_FILE) INTO (INPUT_RECORD); | | |
| 4700 | | | | |
| 4800 | 12 | 1 1 DO WHILE (MORE_RECORDS); | | |
| 4900 | 13 | 1 1 1 RELATIVE_RECORD(REL_INDEX) = INPUT_RECORD; | | |
| 5000 | 14 | 1 1 1 IF REL_INDEX ^= 5 THEN | | |
| 5100 | | 1 1 1 REL_INDEX = REL_INDEX + 1; | | |
| 5200 | 15 | 1 1 1 ELSE | | |
| 5300 | | 1 1 1 DO; | | |

Figure 8-6 (Part 1 of 2). Program Writing to an Arrival Sequence File by RRN

| | | | | | | |
|------------------------|--------|---------------------|-----------|---|-------------------|-----------------|
| 5728PL1 R01 M00 880715 | | PL/I Source Listing | | PLITST/LP1415 | 11/30/88 14:18:54 | Page 3 |
| | | | | LP1415: PROCEDURE; | | PUB00160 |
| Include | SEQNBR | STMT.SUBS | BLK BN DO | *<...1...+...2...+...3...+...4...+...5...+...6...+...7...>...8 Date | | |
| | 5400 | 16 | 1 1 2 | REL_INDEX = 1; | | PUB00680 |
| | 5500 | 17 | 1 1 2 | 12 WRITE FILE (REL_FILE) FROM (RELATIVE_RECORD_01) | | PUB00690 |
| | 5600 | | 1 1 2 | KEYFROM (INPUT_WEEK); | | PUB00700 |
| | 5700 | 18 | 1 1 2 | END; /* DO */ | | PUB00710 |
| | 5800 | 19 | 1 1 1 | READ FILE (IN_FILE) INTO (INPUT_RECORD); | | PUB00720 |
| | 5900 | 20 | 1 1 1 | END; /* DO WHILE */ | | PUB00730 |
| | 6000 | | | | | PUB00740 |
| | 6100 | 21 | 1 1 | CLOSE | | PUB00750 |
| | 6200 | | 1 1 | FILE (IN_FILE); | | PUB00760 |
| | 6300 | 22 | 1 1 | CLOSE | | PUB00770 |
| | 6400 | | 1 1 | FILE (REL_FILE); | | PUB00780 |
| | 6500 | | | | | PUB00790 |
| | 6600 | 23 | 1 1 | END LP1415; | | PUB00800 830817 |

Figure 8-6 (Part 2 of 2). Program Writing to an Arrival Sequence File by RRN

- 1 RECORD data transmission is used with IN_FILE.
- 2 The SEQUENTIAL access method is used with IN_FILE.
- 3 IN_FILE is used for INPUT only; no output is directed to it.
- 4 The CONSECUTIVE option specifies that the file is processed using the arrival sequence access path. Because CONSECUTIVE is the default, it could have been omitted.
- 5 BUFSIZE(23) indicates that the maximum record length is 23 characters. When the file is opened, this is the amount of storage the program allocates in the buffer for a record. If your data is blocked, your program will run faster if you specify BLOCK instead of BUFSIZE; this is explained in Figure 8-4 on page 8-7.
- 6 RECORD data transmission is used with REL_FILE.
- 7 The DIRECT attribute indicates that REL_FILE is accessed non-sequentially.
- 8 REL_FILE is used for OUTPUT only. It does not provide data for the program.
- 9 The CONSECUTIVE option specifies that REL_FILE has an arrival sequence access path. Because CONSECUTIVE is the default, the ENVIRONMENT attribute can be omitted.

The combination of the DIRECT attribute and the CONSECUTIVE option of the ENVIRONMENT attribute indicates that the file is accessed non-sequentially, without using a key field in the record. The record is found by using the relative record number, which indicates how far the record is from the start of the file.

This program does not create an RRN-type file. The file must be created first, by creating a data base file with an arrival sequence access path and placing empty records in the file. The number of empty records placed in the file must be greater than or equal to the highest RRN which will later be used to access the file.

USING DATA BASE FILES

- 10** IN_FILE is opened with the TITLE option specifying 'INFILE'. Because the library and member names are allowed to default, the first member of file IN_FILE in the library list is opened.
- 11** REL_FILE is opened with the TITLE option specifying 'MSTFILE'. Because the library and member names are allowed to default, the first member of file MSTFILE in the library list is opened.
- 12** RELATIVE_RECORD_01 is written to REL_FILE using INPUT_WEEK as the relative record number. Because RELATIVE_RECORD_01 is an array which is filled by five readings of INPUT_RECORD, the assumption is made that INPUT_WEEK in every fifth INPUT_RECORD is the same as in the four preceding records, but that each of these five records has a different INPUT_YEAR.

Example 7 - Updating an Arrival Sequence File by Relative Record Number

| 5728PL1 R01 M00 880715 | PL/I Source Listing | PLITST/LP1416 | 11/30/88 14:20:10 | Page 2 |
|------------------------|---|----------------------------|-------------------|-----------------|
| Include | LP1416: PROCEDURE; | | | PUB00160 |
| 100 | LP1416: PROCEDURE; | | | PUB00160 830817 |
| 200 | | | | PUB00170 |
| 300 | /* FILE DECLARATIONS */ | | | PUB00180 |
| 400 | DECLARE | 1 | 2 | PUB00190 |
| 500 | IN_FILE FILE RECORD INTERNAL SEQUENTIAL INPUT ENV(CONSECUTIVE | | 3 | PUB00200 831004 |
| 600 | | | 4 | 831004 |
| 700 | REL_FILE FILE RECORD INTERNAL DIRECT UPDATE ENV(CONSECUTIVE); | | 5 | PUB00210 |
| 800 | | 6 | 7 | PUB00220 |
| 900 | /* RECORD DECLARATIONS */ | | 8 | PUB00230 |
| 1000 | DECLARE | | | PUB00240 |
| 1100 | 1 RELATIVE_RECORD_01, | | | PUB00250 |
| 1200 | 2 PREVIOUS_WORK_YEARS | CHAR(92), | | PUB00260 831004 |
| 1300 | 2 LATEST_WORK_YEAR | CHAR(23), | | PUB00270 831004 |
| 1400 | 1 INPUT_RECORD, | | | PUB00280 |
| 1500 | 2 INPUT_YEAR | PICTURE '99', | | PUB00290 831114 |
| 1600 | 2 INPUT_WEEK | PICTURE '99', | | PUB00300 831114 |
| 1700 | 2 INPUT_UNIT_SALES | PICTURE '\$999999', | | PUB00310 |
| 1800 | 2 INPUT_DOLLAR_SALES | PICTURE '\$999999999V99', | | PUB00320 |
| 1900 | 1 WORK_RECORD, | | | PUB00330 |
| 2000 | 2 OLDEST_WEEK | CHAR(23), | | PUB00340 831004 |
| 2100 | 2 CURRENT_WORK_YEARS | CHAR(92); | | PUB00350 831004 |
| 2200 | | | | PUB00360 |
| 2300 | /* PROGRAM VARIABLES */ | | | PUB00370 |
| 2400 | DECLARE | | | PUB00380 |
| 2500 | INPUT_RECORD_B | CHAR(23) BASED (P1), | | PUB00390 831004 |
| 2600 | P1 | POINTER, | | PUB00400 |
| 2700 | ADDR | BUILTIN; | | PUB00410 |
| 2800 | | | | PUB00420 |
| 2900 | /* PROGRAM FLAGS */ | | | PUB00430 |
| 3000 | DECLARE | | | PUB00440 |
| 3100 | 1 BIT_FLAGS STATIC, | | | PUB00450 |
| 3200 | 2 MORE_RECORDS | BIT(1) ALIGNED, | | PUB00460 |
| 3300 | 2 NO | BIT(1) ALIGNED INIT('0'B), | | PUB00470 |
| 3400 | 2 YES | BIT(1) ALIGNED INIT('1'B); | | PUB00480 |
| 3500 | | | | PUB00490 |
| 3600 | ON ENDFILE(IN_FILE) | | | PUB00500 |
| 3700 | MORE_RECORDS = NO; | | | PUB00510 |
| 3800 | | | | PUB00520 |
| 3900 | ON ENDFILE(REL_FILE) | | | PUB00530 |
| 4000 | MORE_RECORDS = NO; | | | PUB00540 |
| 4100 | | | | PUB00550 |
| 4200 | /* MAIN PROGRAM */ | | | PUB00560 |
| 4300 | MORE_RECORDS = YES; | | | PUB00570 |
| 4400 | P1 = ADDR(INPUT_RECORD); | | | PUB00580 |
| 4500 | | | | PUB00590 |
| 4600 | 9 OPEN | | | PUB00600 |
| 4700 | FILE (IN_FILE) TITLE('UPDATES'); /* INPUT */ | | | PUB00610 831004 |
| 4800 | 10 OPEN | | | PUB00620 |
| 4900 | FILE (REL_FILE) TITLE('MSTFILE'); /* UPDATE */ | | | PUB00630 831004 |
| 5000 | | | | PUB00640 |
| 5100 | 11 READ FILE (IN_FILE) INTO (INPUT_RECORD); | | | PUB00650 |
| 5200 | READ FILE (REL_FILE) INTO (WORK_RECORD) KEY (INPUT_WEEK); | | | PUB00660 |
| 5300 | | | | PUB00670 |

Figure 8-7 (Part 1 of 2). Program Updating an Arrival Sequence File by RRN

USING DATA BASE FILES

| 5728PL1 R01 M00 880715 | | PL/I Source Listing | | PLITST/LP1416 | 11/30/88 14:20:10 | Page 3 |
|------------------------|--------|---------------------|-----------|--|-------------------|-----------------|
| Include | SEQNBR | STMT.SUBS | BLK BN DO | LP1416: PROCEDURE; *<..+....1....+....2....+....3....+....4....+....5....+....6....+....7>..+....8 Date | | PUB00160 |
| | 5400 | 14 | 1 1 | DO WHILE (MORE_RECORDS); | | PUB00680 |
| | 5500 | 15 | 1 1 1 | PREVIOUS_WORK_YEARS = CURRENT_WORK_YEARS; | | PUB00690 |
| | 5600 | 16 | 1 1 1 | LATEST_WORK_YEAR = INPUT_RECORD_B; | | PUB00700 |
| | 5700 | 17 | 1 1 1 | 12 REWRITE FILE (REL_FILE) FROM (RELATIVE_RECORD_01) KEY (INPUT_WEEK); | | PUB00710 |
| | 5800 | 18 | 1 1 1 | READ FILE (IN_FILE) INTO (INPUT_RECORD); | | PUB00720 |
| | 5900 | 19 | 1 1 1 | READ FILE (REL_FILE) INTO (WORK_RECORD) KEY (INPUT_WEEK); | | PUB00730 |
| | 6000 | 20 | 1 1 1 | END; /* DO WHILE */ | | PUB00740 |
| | 6100 | | | | | PUB00750 |
| | 6200 | 21 | 1 1 | CLOSE | | PUB00760 |
| | 6300 | | 1 1 | FILE (IN_FILE); | | PUB00770 |
| | 6400 | 22 | 1 1 | CLOSE | | PUB00780 |
| | 6500 | | 1 1 | FILE (REL_FILE); | | PUB00790 |
| | 6600 | | | | | PUB00800 |
| | 6700 | 23 | 1 1 | END LP1416; | | PUB00810 830817 |

Figure 8-7 (Part 2 of 2). Program Updating an Arrival Sequence File by RRN

- 1 RECORD data transmission is used with IN_FILE.
- 2 The SEQUENTIAL access method is used with IN_FILE. IN_FILE is used for INPUT only; no data is directed to it.
- 3 The CONSECUTIVE option specifies that the file is processed using the arrival sequence access path. Because CONSECUTIVE is the default, it could have been omitted.
- 4 BUFSIZE(23) indicates that the maximum record length is 23 characters. When the file is opened, this is the amount of storage the program allocates in the buffer for a record. If your data is blocked, your program will run faster if you specify BLOCK instead of BUFSIZE. This is explained at Figure 8-4 on page 8-7.
- 5 RECORD data transmission is used with REL_FILE.
- 6 REL_FILE is declared with the DIRECT attribute, because it will be accessed non-sequentially.
- 7 REL_FILE is declared with the UPDATE attribute, because records in the file are being read, altered, and then rewritten. In Figure 8-6 on page 8-14, the file was declared with the OUTPUT attribute because it was being loaded with initial data.
- 8 The CONSECUTIVE option indicates that REL_FILE is processed using the arrival sequence access path. Although REL_FILE does not have a keyed sequence access path, it can be accessed sequentially by the relative record number.
- 9 IN_FILE is opened, with the TITLE option specifying 'UPDATES'. Since the library and member names are allowed to default, the first member of file UPDATES in the library list is opened.
- 10 REL_FILE is opened, with the TITLE option specifying 'MSTFILE'. Because the library and member names are allowed to default, the first member of file MSTFILE in the library list is opened.

- 11** A record is read in from IN_FILE. A key INPUT_WEEK is obtained from this record and is used as the relative record number to obtain the corresponding record from REL_FILE.
- 12** After the record from REL_FILE has been altered to incorporate the new information for IN_FILE, it is rewritten using the same key with which it was read. The record's position in REL_FILE will stay the same.

USING DATA BASE FILES

Example 8 - Reading from an Arrival Sequence File by Relative Record Number

| 5728PL1 R01 M00 880715 | PL/I Source Listing | PLITST/LP1417 | 11/30/88 14:21:28 | Page 2 |
|------------------------|----------------------------|---|--|-----------------|
| Include | SEQNBR STMT.SUBS BLK BN DO | LP1417: PROCEDURE; LP1417: PROCEDURE; | *<...1...2...3...4...5...6...7...>...8 | Date |
| 100 | 1 | | | PUB00160 |
| 200 | | | | PUB00160 830817 |
| 300 | | /* FILE DECLARATIONS */ | | PUB00170 |
| 400 | 2 | 1 1 DECLARE | | PUB00180 |
| 500 | | 1 1 IN_FILE FILE RECORD INTERNAL SEQUENTIAL INPUT ENV(CONSECUTIVE | | PUB00190 |
| 600 | | 1 1 | | PUB00200 831004 |
| 700 | 2.1 | 1 1 REL_FILE FILE RECORD INTERNAL DIRECT INPUT ENV(CONSECUTIVE), | | 831004 |
| 800 | 2.2 | 1 1 SYSPRINT FILE STREAM OUTPUT PRINT; | | PUB00210 831004 |
| 900 | | | | 830607 |
| 1000 | | /* RECORD DECLARATIONS */ | | PUB00220 |
| 1100 | 3 | 1 1 DECLARE | | PUB00230 |
| 1200 | | 1 RELATIVE_RECORD_01, | | PUB00240 |
| 1300 | 3.1 | 1 1 2 RELATIVE_RECORD(5), | | PUB00250 |
| 1400 | 3.2 | 1 1 3 REL_YEAR | PICTURE '99', | PUB00270 831114 |
| 1500 | 3.3 | 1 1 3 REL_WEEK | PICTURE '99', | PUB00280 831114 |
| 1600 | 3.4 | 1 1 3 REL_UNIT_SALES | PICTURE '\$999999', | PUB00290 |
| 1700 | 3.5 | 1 1 3 REL_DOLLAR_SALES | PICTURE '\$99999999V99', | PUB00300 |
| 1800 | 3.6 | 1 1 1 INPUT_RECORD, | | PUB00310 |
| 1900 | 3.7 | 1 1 2 INPUT_WEEK | PICTURE '99', | PUB00320 831114 |
| 2000 | 3.8 | 1 1 2 END_WEEK | PICTURE '99'; | PUB00330 831114 |
| 2100 | | | | PUB00340 |
| 2200 | | /* PROGRAM FLAGS */ | | PUB00350 |
| 2300 | 4 | 1 1 DECLARE | | PUB00360 |
| 2400 | | 1 BIT_FLAGS STATIC, | | PUB00370 |
| 2500 | 4.1 | 1 1 2 MORE_RECORDS | BIT(1) ALIGNED, | PUB00380 |
| 2600 | 4.2 | 1 1 2 NO | BIT(1) ALIGNED INIT('0'B), | PUB00390 |
| 2700 | 4.3 | 1 1 2 YES | BIT(1) ALIGNED INIT('1'B); | PUB00400 |
| 2800 | | | | PUB00410 |
| 2900 | | /* PROGRAM VARIABLES */ | | PUB00420 |
| 3000 | 5 | 1 1 DECLARE | | PUB00430 |
| 3100 | | 1 SEQ_INCR | BINARY FIXED(2), | PUB00440 |
| 3200 | 5.1 | 1 1 REL_INDEX | BINARY FIXED(2); | PUB00450 |
| 3300 | | | | PUB00460 |
| 3400 | 6 | 1 1 ON ENDFILE(IN_FILE) | | PUB00470 |
| 3500 | | 1 1 MORE_RECORDS = NO; | | PUB00480 |
| 3600 | | | | PUB00490 |
| 3700 | 7 | 1 1 ON KEY(REL_FILE) | | PUB00500 |
| 3800 | | 1 1 BEGIN; | | PUB00510 |
| 3900 | 8 | 3 2 ON ERROR SYSTEM; | | PUB00520 |
| 4000 | 9 | 3 2 REL_WEEK(1) = 53; | | PUB00530 831114 |
| 4100 | 10 | 3 2 END; /* BEGIN */ | | PUB00540 |
| 4200 | | | | PUB00550 |
| 4300 | | /* MAIN PROGRAM */ | | PUB00560 |
| 4400 | 11 | 1 1 MORE_RECORDS = YES; | | PUB00570 |
| 4500 | | | | PUB00580 |
| 4600 | 12 | 1 1 13 OPEN | | PUB00590 |
| 4700 | | 1 1 FILE (IN_FILE) TITLE('RETRIEVE'); /* INPUT */ | | PUB00600 831004 |
| 4800 | 13 | 1 1 14 OPEN | | PUB00610 |
| 4900 | | 1 1 FILE (REL_FILE) TITLE('MSTFILE'); /* INPUT */ | | PUB00620 831004 |
| 5000 | | | | PUB00630 |
| 5100 | 14 | 1 1 READ FILE (IN_FILE) INTO (INPUT_RECORD); | | PUB00640 |
| 5200 | | | | PUB00650 |
| 5300 | 15 | 1 1 DO WHILE (MORE_RECORDS); | | PUB00660 |

Figure 8-8 (Part 1 of 2). Program Reading from an Arrival Sequence File by RRN

| 5728PL1 R01 M00 880715 | PL/I Source Listing | PLITST/LP1417 | 11/30/88 14:21:28 | Page 3 |
|------------------------|----------------------------|--|-------------------|-----------------|
| Include | SEQNBR STMT.SUBS BLK BN DO | *<..+...1...+...2...+...3...+...4...+...5...+...6...+...7...>...+...8 Date | | |
| | 5400 16 1 1 1 | LP1417: PROCEDURE; | | PUB00160 |
| | | IF END_WEEK = 00 THEN | | PUB00670 831114 |
| | 5500 1 1 1 | CALL RANDOM; | | PUB00680 |
| | 5600 17 1 1 1 | ELSE | | PUB00690 |
| | 5700 1 1 1 | CALL SEQUENT; | | PUB00700 |
| | 5800 18 1 1 1 | READ FILE (IN_FILE) INTO (INPUT_RECORD); | | PUB00710 |
| | 5900 19 1 1 1 | END; /* DO WHILE */ | | PUB00720 |
| | 6000 | | | PUB00730 |
| | 6100 20 1 1 | CLOSE | | PUB00740 |
| | 6200 1 1 | FILE (IN_FILE); | | PUB00750 |
| | 6300 21 1 1 | CLOSE | | PUB00760 |
| | 6400 1 1 | FILE (REL_FILE); | | PUB00770 |
| | 6500 | | | PUB00780 |
| | 6600 22 1 1 | RANDOM: PROCEDURE; | | PUB00790 |
| | 6700 23 4 2 | 15 READ FILE (REL_FILE) INTO (RELATIVE_RECORD_01) KEY (INPUT_WEEK); | | PUB00800 |
| | 6800 24 4 2 | IF REL_WEEK(1) = 53 THEN | | PUB00810 831114 |
| | 6900 4 2 | DO REL_INDEX = 1 TO 5; | | PUB00820 |
| | 7000 25 4 2 1 | CALL PRT_SMY; | | PUB00830 |
| | 7100 26 4 2 1 | END; /* DO LOOP */ | | PUB00840 |
| | 7200 27 4 2 | END RANDOM; | | PUB00850 |
| | 7300 | | | PUB00860 |
| | 7400 28 1 1 | SEQUENT: PROCEDURE; | | PUB00870 |
| | 7500 29 5 2 | SEQ_INCR = 1; | | PUB00880 |
| | 7600 30 5 2 | 16 READ FILE (REL_FILE) INTO (RELATIVE_RECORD_01) KEY (INPUT_WEEK); | | PUB00890 |
| | 7700 31 5 2 | IF REL_WEEK(1) = 53 THEN | | PUB00900 831114 |
| | 7800 5 2 | DO WHILE (REL_WEEK(1) <= END_WEEK); | | PUB00910 |
| | 7900 32 5 2 1 | DO REL_INDEX = 1 TO 5; | | PUB00920 |
| | 8000 33 5 2 2 | CALL PRT_SMY; | | PUB00930 |
| | 8100 34 5 2 2 | END; /* DO LOOP */ | | PUB00940 |
| | 8200 35 5 2 1 | 17 READ FILE (REL_FILE) INTO (RELATIVE_RECORD_01) | | PUB00950 |
| | 8300 5 2 1 | KEY (INPUT_WEEK + SEQ_INCR); | | PUB00960 831114 |
| | 8400 36 5 2 1 | SEQ_INCR = SEQ_INCR + 1; | | PUB00970 |
| | 8500 37 5 2 1 | END; /* DO WHILE */ | | PUB00980 |
| | 8600 38 5 2 | END SEQUENT; | | PUB00990 |
| | 8700 | | | PUB01000 |
| | 8800 39 1 1 | PRT_SMY: PROCEDURE; | | PUB01010 |
| | 8900 40 6 2 | 18 PUT FILE (SYSPRINT) SKIP(2) EDIT(REL_YEAR(REL_INDEX), | | PUB01020 |
| | 9000 6 2 | REL_WEEK(REL_INDEX),REL_UNIT_SALES(REL_INDEX), | | PUB01030 |
| | 9100 6 2 | REL_DOLLAR_SALES(REL_INDEX))(A(2),X(5),A(2),X(5), | | PUB01040 831004 |
| | 9200 6 2 | F(8),X(5),F(14,2)); | | PUB01050 831004 |
| | 9300 41 6 2 | END PRT_SMY; | | PUB01060 |
| | 9400 | | | PUB01070 |
| | 9500 42 1 1 | END LP1417; | | PUB01080 830817 |

Figure 8-8 (Part 2 of 2). Program Reading from an Arrival Sequence File by RRN

- 1** RECORD data transmission is used with IN_FILE.
- 2** The SEQUENTIAL access method is used with IN_FILE.
- 3** IN_FILE is used for INPUT only; no data is directed to it.
- 4** The CONSECUTIVE option specifies that the file is processed using the arrival sequence access path. Because CONSECUTIVE is the default, it could have been omitted.
- 5** BUFSIZE(4) indicates that the maximum record length is four characters. When the file is opened, this is the amount of storage the program allocates in the buffer for a record. If your data is blocked, the program will run faster if you specify BLOCK instead of BUFSIZE; this is explained at note 5 of Figure 8-4 on page 8-7.
- 6** RECORD data transmission is used with REL_FILE.

USING DISPLAY FILES

- 7** REL_FILE is declared with the DIRECT attribute because it is accessed non-sequentially.
- 8** REL_FILE is used for INPUT only; no data is directed to it.
- 9** The CONSECUTIVE option specifies that REL_FILE has an arrival sequence access path. Because CONSECUTIVE is the default, the ENVIRONMENT attribute can be omitted.
- 10** STREAM data transmission is used with SYSPRINT. Stream files can only be accessed sequentially.
- 11** The file receives OUTPUT only. It does not supply data to the program.
- 12** The PRINT attribute indicates that the first character in the record is an ASA printer control character.
- 13** IN_FILE is opened, with the TITLE option specifying 'RETRIEVE'. Because the library and member names are allowed to default, the first member of file RETRIEVE in the library list is opened.
- 14** REL_FILE is opened, with the TITLE option specifying 'MSTFILE'. Because the library and member names are allowed to default, the first member of file MSTFILE in the library list is opened.
- 15** If the input data calls for the retrieval and printing of only one relative record, the READ is done using INPUT_WEEK as the relative record number.
- 16** If the input data calls for the retrieval and printing of several relative records, the first relative record is retrieved using INPUT_WEEK as relative record number.
- 17** For later READ statements an expression (INPUT_WEEK + SEQ_INCR) is used to specify each successive relative record number, until the last record asked for has been retrieved.
- 18** SYSPRINT is implicitly opened by the first processing of the PUT statement. Any file not already opened is implicitly opened by the first data transmission statement which accesses it. It is, however, a good practice to explicitly open all your files.

Using Display Files

When using a display file, there are two file organizations you can specify by using the ENVIRONMENT options CONSECUTIVE and INTERACTIVE. There are also two file access methods you can specify by using the DECLARE statement attributes SEQUENTIAL and SEQUENTIAL KEYED. The input/output statement options allowed with each combination of these are shown in Appendix C, "Valid Combinations of Options for Input/Output Statements." The combination of INTERACTIVE SEQUENTIAL KEYED is mainly used for subfiles. Subfiles are described in *Programming: Data Management Guide*.

Other ENVIRONMENT options you specify can affect the input/output statement options you are able to use. If you specify NOINDARA, you cannot specify the

OPTIONS parameter INDICATORS, and if you specify BUFSIZE, the SET option of the input/output statements is affected. ENVIRONMENT parameters and the OPTIONS option are discussed in Chapter 7, "File Declaration and Input/Output," while other input/output statement options are discussed in "Record Data Transmission" on page 11-8.

Externally Described Display Files

Just as with externally described data base files, you can specify the ENVIRONMENT option DESCRIBED with externally described display files. A description of this option is in Chapter 7, "File Declaration and Input/Output," while level checking is discussed in Chapter 6, "AS/400 PL/I File and Record Management."

Example of Using a Display File

The following program uses a display device file to process customer inquiries.

| File | | Keying Instruction | Graphic Key | Description | Page of |
|------------|------|--------------------|-------------|-------------|---------|
| Programmer | Date | | | | |

| Sequence Number | Form Type | Conditioning | Name | Length | Reference ID | Location | Functions |
|-----------------|-----------|--------------|------------------------------|--------|--------------|---|-----------|
| | | | | | | | |
| 1 | A | | CUSTOMER MASTER INQUIRY FILE | | | -- CUSMINQ | |
| 2 | A | | | | | INDARA | |
| 3 | A | | | | | PRINT | |
| 4 | A | | | | | REF(CUSMSTP) | |
| 5 | A | | R CUSPMT | | | CA01(01 'END OF PROGRAM') | |
| 6 | A | | | | | TEXT('CUSTOMER PROMPT') | |
| 7 | A | | | | | 1 3 'CUSTOMER MASTER INQUIRY' | |
| 8 | A | | | | | 3 'CUSTOMER NUMBER' | |
| 9 | A | 99 | CUST | R | | 1 3 20 | |
| 10 | A | | | | | ERRMSG('CUSTOMER NUMBER NOT FOUND: + | |
| 11 | A | | | | | PRESS RESET, THEN ENTER VALID NUMBE+ | |
| 12 | A | | | | | R' 99) | |
| 13 | A | | | | | 5 3 'USE F1 TO END PROGRAM, USE ENTER + | |
| 14 | A | | | | | KEY TO RETURN TO PROMPT SCREEN' | |
| 15 | A | | R CUSFLDS | | | TEXT('CUSTOMER DISPLAY') | |
| 16 | A | | | | | OVERLAY | |
| 17 | A | | | | | 8 3 'NAME' | |

Figure 8-9 (Part 1 of 5). Display File DDS and Customer Inquiry Program

USING DISPLAY FILES

IBM International Business Machines

AS/400 DATA DESCRIPTION SPECIFICATIONS

G121-8891-0 UM/050-
Printed in U.S.A.
*Number of sheets per pad may vary slightly.

| File | | Keying Instruction | Graphic | | Description | Page of |
|------------|------|-----------------------|---------|--|-------------|------------|
| Programmer | Date | | Key | | | |

| Sequence Number | Event Type 1-3 4-6 7-9 10-12 13-15 16-18 19-21 22-24 25-27 28-30 31-33 34-36 37-39 40-42 43-45 46-48 49-51 52-54 55-57 58-60 61-63 64-66 67-69 70-72 73-75 76-78 79-81 | Conditioning | | | Name | Length | Location | | Functions |
|--------------------|---|--------------|-----------|-----------|-------|--------|----------|-------|----------------------|
| | | Indicator | Indicator | Indicator | | | Line | Pos | |
| A | | | | | NAME | R | B | 8 11 | |
| A | | | | | ADDR | R | B | 9 11 | 'ADDRESS' |
| A | | | | | CITY | R | B | 10 11 | 'CITY' |
| A | | | | | STATE | R | B | 11 11 | 'STATE' |
| A | | | | | ZIP | R | B | 11 21 | 'ZIP CODE' |
| A | | | | | ARBAL | R | B | 12 17 | 'A/R BALANCE' (J) |
| A | | | | | | | | | |
| A | | | | | | | | | |
| A | | | | | | | | | |
| A | | | | | | | | | |
| A | | | | | | | | | |
| A | | | | | | | | | |
| A | | | | | | | | | |
| A | | | | | | | | | |

Figure 8-9 (Part 2 of 5). Display File DDS and Customer Inquiry Program

USING DISPLAY FILES

```

5728PL1 R01 M00 880715          PL/I Source Listing          LP1420/LP1420          11/30/87 15:35:35          Page 2
LP1420: PROCEDURE;
LP1420: PROCEDURE;
Include  SEQNBR STMT.SUBS BLK BN DO *<.....1.....2.....3.....4.....5.....6.....7.....8 Date
100      1
200
300      /* FILE DECLARATIONS */
400      2      1 1      DECLARE
500      1 1      CUSMINQ FILE RECORD SEQUENTIAL UPDATE ENV(INTERACTIVE),
600      2.1    1 1      CUSMSTP FILE RECORD DIRECT INPUT ENV(INDEXED DESCRIBED);
700      5      6      7      8      9
800      /* RECORD DECLARATIONS */
900      3      1 1      DECLARE
1000     1 1      1 CUSTOMER_PROMPT,
1100     %INCLUDE CUSMINQ(CUSPMT,INPUT,,COMMA);
CUSPMT   + 100      /* ----- */
CUSPMT   + 200      /* DEVICE FILE: CUSMINQ.LP1420 */
CUSPMT   + 300      /* FILE CREATION DATE: 87/11/30 */
CUSPMT   + 400      /* RECORD FORMAT: CUSPMT */
CUSPMT   + 500      /* RECORD FORMAT SEQUENCE ID: 12E4847185567 */
CUSPMT   + 600      /* ----- */
CUSPMT   + 700      /* INDICATORS FOR FORMAT CUSPMT */
CUSPMT   + 800      /* INDICATOR 03      End of Program */
CUSPMT   + 900      /* INDICATOR 99      Customer number not found: press Reset, then
CUSPMT   + 1000     enter
CUSPMT   + 1100     /* -----Customer Prompt----- */
CUSPMT   + 1200     3.1  1 1      15 CUST      CHAR(5),      /* CUSTOMER NUMBER
CUSPMT   + 1200     3.2  1 1      1 CUSTOMER_INDICATORS,
CUSPMT   + 1300     %INCLUDE CUSMINQ(CUSPMT,INDICATORS,,COMMA);
CUSPMT   + 100      /* ----- */
CUSPMT   + 200      /* DEVICE FILE: CUSMINQ.LP1420 */
CUSPMT   + 300      /* FILE CREATION DATE: 87/11/30 */
CUSPMT   + 400      /* RECORD FORMAT: CUSPMT */
CUSPMT   + 500      /* RECORD FORMAT SEQUENCE ID: 12E4847185567 */
CUSPMT   + 600      /* -----Customer Prompt----- */
CUSPMT   + 700     3.3  1 1      15 IN01      PIC '9',      /* End of Program
CUSPMT   + 800     3.4  1 1      15 IN02_IN98 CHAR(97),      /* UNDEFINED INDICATOR(S)
CUSPMT   + 900     3.5  1 1      15 IN99      PIC '9',      /* Customer number not found:
CUSPMT   + 1000     press Reset, then enter
CUSPMT   + 1400     3.6  1 1      1 CUSTOMER_FIELDS,
CUSPMT   + 1500     %INCLUDE CUSMINQ(CUSFLDS,OUTPUT,,COMMA);
CUSFLDS  + 100      /* ----- */
CUSFLDS  + 200      /* DEVICE FILE: CUSMINQ.LP1420 */
CUSFLDS  + 300      /* FILE CREATION DATE: 87/11/30 */
CUSFLDS  + 400      /* RECORD FORMAT: CUSFLDS */
CUSFLDS  + 500      /* RECORD FORMAT SEQUENCE ID: 1B8CFB695D44B */
CUSFLDS  + 600      /* ----- */
CUSFLDS  + 700      /* INDICATORS FOR FORMAT CUSFLDS */
CUSFLDS  + 800      /* INDICATOR 03      End of Program */
CUSFLDS  + 900      /* -----Customer Display----- */
CUSFLDS  + 1000     3.7  1 1      15 NAME      CHAR(25),      /* CUSTOMER NAME
CUSFLDS  + 1100     3.8  1 1      15 ADDR      CHAR(20),      /* CUSTOMER ADDRESS
CUSFLDS  + 1200     3.9  1 1      15 CITY      CHAR(20),      /* CUSTOMER CITY
CUSFLDS  + 1300     3.10 1 1     15 STATE     CHAR(2),      /* STATE
CUSFLDS  + 1400     3.11 1 1     15 ZIP       PIC '9999R',      /* ZIP CODE
CUSFLDS  + 1500     3.12 1 1     15 ARBAL     PIC '999999V9R', /* ACCTS REC BALANCE
CUSFLDS  + 1600     3.13 1 1     1 CUSTOMER_MASTER_RECORD,

```

Figure 8-9 (Part 3 of 5). Display File DDS and Customer Inquiry Program

USING DISPLAY FILES

```

5728PL1 R01 M00 880715          PL/I Source Listing          LP1420/LP1420          11/30/88 15:35:35          Page 3
                                LP1420: PROCEDURE;          DIS00160
Include      SEQNBR STMT.SUBS BLK BN DO *<.....1.....2.....3.....4.....5.....6.....7.....8 Date
                                %INCLUDE CUSMSTP(CUSMST,RECORD);          DIS00300
CUSMST      + 100                                /* ----- */
CUSMST      + 200                                /* PHYSICAL FILE: CUSMSTP.LP1420 */
CUSMST      + 300                                /* FILE CREATION DATE: 87/11/30 */
CUSMST      + 400                                /* RECORD FORMAT: CUSMST */
CUSMST      + 500                                /* RECORD FORMAT SEQUENCE ID: 4AA7C90E9BB1E */
CUSMST      + 600                                /* -----CUSTOMER MASTER RECORD----- */
CUSMST      + 700      3.14      1 1              15 CUST      CHAR(5),          /* CUSTOMER NUMBER */
CUSMST      + 800                                /* DDS - KEY FIELD */
CUSMST      + 900      3.15      1 1              15 NAME      CHAR(25),          /* CUSTOMER NAME */
CUSMST      + 1000     3.16      1 1              15 ADDR      CHAR(20),          /* CUSTOMER ADDRESS */
CUSMST      + 1100     3.17      1 1              15 CITY      CHAR(20),          /* CUSTOMER CITY */
CUSMST      + 1200     3.18      1 1              15 STATE     CHAR(2),          /* STATE */
CUSMST      + 1300     3.19      1 1              15 ZIP       PIC '9999R',          /* ZIP CODE */
CUSMST      + 1400     3.20      1 1              15 SRHCOD    CHAR(6),          /* CUSTOMER NUMBER SEARCH CODE */
CUSMST      + 1500     3.21      1 1              15 CUSTYP    PIC 'R',          /* CUSTOMER TYPE 1=60V 2=SCH
CUSMST      + 1600                                /* 3=BUS 4=PVT 5=OT */
CUSMST      + 1700     3.22      1 1              15 ARBAL     PIC '999999V9R', /* ACCTS REC BALANCE */
CUSMST      + 1800     3.23      1 1              15 ORDBAL    PIC '999999V9R', /* A/R AMT IN ORDER FILE */
CUSMST      + 1900     3.24      1 1              15 LSTAMT    PIC '999999V9R', /* LAST AMT PAID IN A/R */
CUSMST      + 2000     3.25      1 1              15 LSTDAT    PIC '999999R', /* LAST DATE PAID IN A/R */
CUSMST      + 2100     3.26      1 1              15 CRDLMT    PIC '999999V9R', /* CUSTOMER CREDIT LIMIT */
CUSMST      + 2200     3.27      1 1              15 SLSYR     PIC '99999999V9R',
CUSMST      + 2300                                /* CUSTOMER SALES THIS YEAR */
CUSMST      + 2400     3.28      1 1              15 SLSLYR    PIC '99999999V9R';
CUSMST      + 2500                                /* CUSTOMER SALES LAST YEAR */
1800
1900
2000      4      1 1              /* INDICATOR FLAGS */
2100                                DECLARE
2200      4.1      1 1              1 INDICATOR_FLAGS STATIC,
2300      4.2      1 1              2 OFF          PICTURE '9' INIT(0),
2400                                2 ON          PICTURE '9' INIT(1);
2500
2600      5      1 1              /* BUILT-IN FUNCTIONS */
2700      1 1              DECLARE
2800                                ONCODE          BUILTIN;
2900
2900      6      1 1              ON KEY (CUSMSTP)
3000      1 1              BEGIN;
3100      7      2 2              ON ERROR SYSTEM;
3200      8      2 2              IF ONCODE = 51 THEN
3300      2 2              IN99 = ON;
3400      9      2 2              END; /* BEGIN */
3500
3600
3700      10     1 1              /* MAIN PROGRAM */
3800      1 1              OPEN 10
3900      11     1 1              FILE (CUSMINQ); /* UPDATE */
4000      1 1              OPEN 11
4100      1 1              FILE (CUSMSTP); /* INPUT */
4200      12     1 1              IN03 = OFF;
4300      13     1 1              IN99 = OFF;
4400      14     1 1              DO WHILE(IN03 = OFF);

```

Figure 8-9 (Part 4 of 5). Display File DDS and Customer Inquiry Program

USING DISPLAY FILES

| 5728PL1 R01 M00 880715 | PL/I Source Listing | LP1420/LP1420 | 11/30/88 15:35:35 | Page 4 |
|------------------------|----------------------------|---|-------------------|-----------------|
| | LP1420: PROCEDURE; | | | DIS00160 |
| Include | SEQNBR STMT.SUBS BLK BN DO | *<..+....1....+....2....+....3....+....4....+....5....+....6....+....7.>..+....8 Date | | |
| | 4500 | | | DIS00580 |
| | 4600 | /* DISPLAY THE SCREEN */ | | DIS00590 |
| | 4700 15 | 1 1 1 12 WRITE FILE (CUSMINQ) FROM (CUSTOMER_PROMPT) | | DIS00600 |
| | 4800 | 1 1 1 OPTIONS (RECORD('CUSPMT') INDICATORS(CUSTOMER_INDICATORS)); | | DIS00610 831006 |
| | 4900 | | 13 14 | DIS00620 |
| | 5000 | /* READ AND PROCESS SCREEN */ | | DIS00630 |
| | 5100 16 | 1 1 1 IN99 = OFF; | | DIS00640 |
| | 5200 17 | 1 1 1 15 READ FILE (CUSMINQ) INTO (CUSTOMER_PROMPT) | | DIS00650 |
| | 5300 | 1 1 1 OPTIONS (RECORD('CUSPMT') INDICATORS(CUSTOMER_INDICATORS)); | | DIS00660 831006 |
| | 5400 18 | 1 1 1 16 IF IN01 = OFF THEN 17 | | DIS00670 |
| | 5500 | 1 1 1 DO; | | DIS00680 |
| | 5600 19 | 1 1 2 READ FILE (CUSMSTP) INTO (CUSTOMER_MASTER_RECORD) | | DIS00690 831006 |
| | 5700 | 1 1 2 KEY(CUSTOMER_PROMPT.CUST); | | 831006 |
| | 5800 20 | 1 1 2 19 IF IN99 = OFF THEN | | DIS00700 |
| | 5900 | 1 1 2 DO; | | DIS00710 |
| | 6000 21 | 1 1 3 CUSTOMER_FIELDS = CUSTOMER_MASTER_RECORD, BY NAME; | | DIS00720 |
| | 6100 22 | 1 1 3 WRITE FILE (CUSMINQ) FROM (CUSTOMER_FIELDS) | | DIS00730 |
| | 6200 | 1 1 3 OPTIONS(RECORD('CUSFLDS')); | | DIS00740 |
| | 6300 23 | 1 1 3 20 READ FILE (CUSMINQ) INTO (CUSTOMER_FIELDS) | | DIS00750 |
| | 6400 | 1 1 3 OPTIONS(RECORD('CUSFLDS') | | DIS00760 831006 |
| | 6500 | 1 1 3 INDICATORS(CUSTOMER_INDICATORS)); | | 831006 |
| | 6600 24 | 1 1 3 END; /* DO */ | | DIS00770 |
| | 6700 25 | 1 1 2 END; /* DO */ | | DIS00780 |
| | 6800 26 | 1 1 1 END; /* DO WHILE */ | | DIS00790 |
| | 6900 | | | DIS00800 |
| | 7000 27 | 1 1 CLOSE | | DIS00810 |
| | 7100 | 1 1 FILE (CUSMINQ); | | DIS00820 |
| | 7200 28 | 1 1 CLOSE | | DIS00830 |
| | 7300 | 1 1 FILE (CUSMSTP); | | DIS00840 |
| | 7400 | | | DIS00850 |
| | 7500 29 | 1 1 END LP1420; | | DIS00860 830817 |

Figure 8-9 (Part 5 of 5). Display File DDS and Customer Inquiry Program

CUSTOMER MASTER INQUIRY

CUSTOMER NUMBER 12345

USE F3 TO END PROGRAM, USE ENTER KEY TO RETURN TO PROMPT SCREEN

NAME EVANS, T.J.

ADDRESS 85 NOWHERE RD.

CITY SCARBERIA

STATE ON ZIP CODE 76889

A/R BALANCE 111,111.11

Figure 8-10. Display Produced Using Display File DDS and Customer Inquiry Program

1 RECORD data transmission is used with CUSMINQ.

USING DISPLAY FILES

- 2** The SEQUENTIAL access method is used with CUSMINQ. The records are read in using the arrival sequence access path.
- 3** The data transmission mode is specified as UPDATE, since CUSMINQ is used for both input and output.
- 4** The ENVIRONMENT attribute is specified with the INTERACTIVE option. Both READ and WRITE statements are allowed, in this case to provide prompt screens and read input which is entered on the work station screen.
- 5** RECORD data transmission is used with CUSMSTP. The DDS for CUSMSTP will be found at Figure 8-1 on page 8-3.
- 6** CUSMSTP is declared with the DIRECT attribute, because it is accessed non-sequentially.
- 7** CUSMSTP is used for INPUT only: no output is directed to it by the program.
- 8** INDEXED specifies that the file is processed using the keyed sequence access path.
- 9** DESCRIBED indicates that external record format definitions are used in the program.
- 10** CUSMINQ is opened. The UPDATE option is omitted, because it is specified as an attribute in the file declaration.
- 11** CUSMSTP is opened. The INPUT option is omitted, because it is specified as an attribute in the file declaration.
- 12** The prompt screen is displayed by writing to CUSMINQ, the file associated with the display device.
- 13** The record format named CUSPMT is used.
- 14** The indicators in CUSTOMER_INDICATORS, a structure declared using the %INCLUDE statement, are passed to the display device.
- 15** The information entered on the work station screen is read into CUSTOMER_PROMPT.
- 16** The record format named CUSPMT is used.
- 17** The indicators are returned from the display device to the structure CUSTOMER_INDICATORS.
- 18** If IN01 has a value of zero, that is, if the user has not requested that the program end, CUSTOMER_MASTER_RECORD is read in from file CUSMSTP using the key the user has provided (CUSTOMER_PROMPT.CUST).
- 19** If IN99 has a value of zero, that is, if the user has specified a valid key, then the necessary fields from CUSTOMER_MASTER_RECORD are placed in CUSTOMER_FIELDS, which is then written to the display device using record format CUSFLDS. If IN99 has a value of one, control is passed to the top of the loop (statement 15) and the prompt screen displays the error

message specified on the DDS reporting that the customer number has not been found, and requesting a valid number for input.

- 20 CUSTOMER_FIELDS is read back in from CUSMINQ, and the indicators are returned from the display device to the structure CUSTOMER_INDICATORS. If the user requests that the program end, the next test of the value of IN01 halts running.

Example of Using a Subfile for Displaying Data

| File | | Keying Instruction | | Graphic Key | | Description | | Page of | |
|------------|------|--------------------|--|-------------|--|-------------|--|---------|--|
| Programmer | Date | | | | | | | | |

| Sequence Number | From Type A M P S D E F G H I J K L N O P Q R S T U V W X Y Z * ^ _ ` ~ / | Conditioning | | | | | Name | Length | Data Type B C D E F G H I J K L M N O P Q R S T U V W X Y Z * ^ _ ` ~ / | Location Lim Pos | Functions |
|-----------------|---|--------------|-----------|-----------|-----------|-----------|---------|--------|--|------------------------|--------------------------------------|
| | | Indicator | Indicator | Indicator | Indicator | Indicator | | | | | |
| A | * | | | | | | ORD220D | | | | EXISTING ORDER REVIEW |
| A | * | | | | | | | | | | INDARA |
| A | * | | | | | | | | | | PRINT |
| A | * | | | | | R | SUB1 | | | | SFL |
| A | * | | | | | | ITEM | 5S | 0 | 10 | 2TEXT('ITEM NUMBER') |
| A | * | | | | | | QTYORD | 3Y | 0 | 10 | 8TEXT('QUANTITY ORDERED') |
| A | * | | | | | | | | | | EDTCDE(3) |
| A | * | | | | | | DESCRP | 30 | | 10 | 14TEXT('ITEM DESCRIPTION') |
| A | * | | | | | | PRICE | 6Y | 2 | 10 | 46TEXT('SELLING PRICE') |
| A | * | | | | | | | | | | EDTCDE(J) |
| A | * | | | | | | EXTENS | 8Y | 2 | 10 | 56EDTCDE(J) |
| A | * | | | | | | | | | | TEXT('EXTENSION AMOUNT OF QTYORD * + |
| A | * | | | | | | | | | | PRICE') |
| A | * | | | | | R | SUBCTL1 | | | | SFLCTL(SUB1) |
| A | * | | | | | | | | | | SFLCLR |
| A | * | | | | | | | | | | SFLDSP |
| A | * | | | | | | | | | | SFLDSPCTL |

Figure 8-11 (Part 1 of 15). Program and Supporting DDS for Using Subfiles

| File | | Keying Instruction | Graphic Key | Description | Page of |
|------------|------|--------------------|-------------|-------------|---------|
| Programmer | Date | | | | |

| Sequence Number | Event Type | Conditioning | | | | | Name | Length | Reference (R) | Location | | Function |
|-----------------|------------|-------------------------|----------|-----------|----------|-----------|------|--------|---------------|----------|-----|---|
| | | Alt/Off/Comment (A/O/?) | Next (N) | Indicator | Next (N) | Indicator | | | | Line | Pos | |
| A | | | | | | | | | | 5 | 44 | 'SHIP VIA' |
| A | | | | | | SHPVIA | 15 | | | 5 | 49 | TEXT('SHIPPING INSTRUCTIONS') |
| A | | | | | | | | | | 6 | 44 | 'PRINTED DATE' |
| A | | | | | | PRDAT | 6Y 0 | | | 6 | 57 | TEXT('DATE ORDER WAS PRINTED') EDTCDE(Y) |
| A | | | | | | | | | | 7 | 29 | 'INVOICE' |
| A | | | | | | INVNUM | 5S 0 | | | 7 | 38 | TEXT('INVOICE NUMBER') |
| A | | | | | | | | | | 7 | 64 | 'MTH' |
| A | | | | | | ACTMTH | 2S 0 | | | 7 | 68 | TEXT('ACCOUNTING MONTH OF SALE') |
| A | | | | | | | | | | 7 | 72 | 'YEAR' |
| A | | | | | | ACTYR | 2S 0 | | | 7 | 77 | TEXT('ACCOUNT YEAR OF SALE') |
| A | | | | | | | | | | 8 | 2 | 'ITEM' |
| A | | | | | | | | | | 8 | 9 | 'QTY' |
| A | | | | | | | | | | 8 | 14 | 'ITEM DESCRIPTION' |
| A | | | | | | | | | | 8 | 48 | 'PRICE' |
| A | | | | | | | | | | 8 | 59 | 'EXTENSION' |

Figure 8-11 (Part 4 of 15). Program and Supporting DDS for Using Subfiles

| File | | Keying Instruction | Graphic Key | Description | Page of |
|------------|------|--------------------|-------------|-------------|---------|
| Programmer | Date | | | | |

| Sequence Number | Event Type | Conditioning | | | | | Name | Length | Reference (R) | Location | | Function |
|-----------------|------------|-------------------------|----------|-----------|----------|-----------|------|--------|---------------|----------|-----|---|
| | | Alt/Off/Comment (A/O/?) | Next (N) | Indicator | Next (N) | Indicator | | | | Line | Pos | |
| A | | | | | | PHYSICAL | | | | | | ORDER HEADER FILE |
| A | | | | | | ORDHDR | | | | | | TEXT('ORDER HEADER RECORD') |
| A | | | | | | CUST | 5 | | | | | TEXT('CUSTOMER NUMBER') |
| A | | | | | | ORDER | 5S 0 | | | | | TEXT('ORDER NUMBER') |
| A | | | | | | ORDDAT | 6S 0 | | | | | TEXT('DATE ORDER WAS ENTERED') |
| A | | | | | | CUSORD | 15 | | | | | TEXT('CUSTOMER PURCHASE ORDER + NUMBER') |
| A | | | | | | SHPVIA | 15 | | | | | TEXT('SHIPPING INSTRUCTIONS') |
| A | | | | | | ORDSTS | 1S 0 | | | | | TEXT('ORDER STATUS 1-PCS 2-CNT + 3-CHK 4-RDY 5-PRT 6-PCK') |
| A | | | | | | OPRNAM | 1S 0 | | | | | TEXT('OPERATOR NAME WHO ENTERED + THE ORDER') |
| A | | | | | | ORDAMT | 8S 2 | | | | | TEXT('TOTAL DOLLAR AMOUNT OF + THE ORDER') |
| A | | | | | | CUSTYP | 1S 0 | | | | | TEXT('CUSTOMER TYPE 1-GOV 2-SCH + 3-BUS 4-PVT 5-OTH') |
| A | | | | | | INVNUM | 5S 0 | | | | | TEXT('INVOICE NUMBER') |
| A | | | | | | PRDAT | 6S 0 | | | | | TEXT('DATE ORDER WAS PRINTED') |
| A | | | | | | OPNSTS | 1S 0 | | | | | TEXT('ORDER OPEN STATUS 1-OPEN + 2-CLOSE 3-CANCEL') |

Figure 8-11 (Part 5 of 15). Program and Supporting DDS for Using Subfiles

USING DISPLAY FILES

IBM International Business Machines

AS/400 DATA DESCRIPTION SPECIFICATIONS

G121-0001-0 UN/050-

Printed in U.S.A.

*Number of sheets per pad may vary slightly.

| File | | Keying Instruction | | Graphic | | Description | | Page of | |
|------------|------|--------------------|------|---------|------|-------------|------|---------|------|
| Programmer | Date | Instruction | Key | Key | Key | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 |
| 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 |
| 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 |
| 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 |
| 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 100 |
| 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 |
| 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 |
| 121 | 122 | 123 | 124 | 125 | 126 | 127 | 128 | 129 | 130 |
| 131 | 132 | 133 | 134 | 135 | 136 | 137 | 138 | 139 | 140 |
| 141 | 142 | 143 | 144 | 145 | 146 | 147 | 148 | 149 | 150 |
| 151 | 152 | 153 | 154 | 155 | 156 | 157 | 158 | 159 | 160 |
| 161 | 162 | 163 | 164 | 165 | 166 | 167 | 168 | 169 | 170 |
| 171 | 172 | 173 | 174 | 175 | 176 | 177 | 178 | 179 | 180 |
| 181 | 182 | 183 | 184 | 185 | 186 | 187 | 188 | 189 | 190 |
| 191 | 192 | 193 | 194 | 195 | 196 | 197 | 198 | 199 | 200 |
| 201 | 202 | 203 | 204 | 205 | 206 | 207 | 208 | 209 | 210 |
| 211 | 212 | 213 | 214 | 215 | 216 | 217 | 218 | 219 | 220 |
| 221 | 222 | 223 | 224 | 225 | 226 | 227 | 228 | 229 | 230 |
| 231 | 232 | 233 | 234 | 235 | 236 | 237 | 238 | 239 | 240 |
| 241 | 242 | 243 | 244 | 245 | 246 | 247 | 248 | 249 | 250 |
| 251 | 252 | 253 | 254 | 255 | 256 | 257 | 258 | 259 | 260 |
| 261 | 262 | 263 | 264 | 265 | 266 | 267 | 268 | 269 | 270 |
| 271 | 272 | 273 | 274 | 275 | 276 | 277 | 278 | 279 | 280 |
| 281 | 282 | 283 | 284 | 285 | 286 | 287 | 288 | 289 | 290 |
| 291 | 292 | 293 | 294 | 295 | 296 | 297 | 298 | 299 | 300 |
| 301 | 302 | 303 | 304 | 305 | 306 | 307 | 308 | 309 | 310 |
| 311 | 312 | 313 | 314 | 315 | 316 | 317 | 318 | 319 | 320 |
| 321 | 322 | 323 | 324 | 325 | 326 | 327 | 328 | 329 | 330 |
| 331 | 332 | 333 | 334 | 335 | 336 | 337 | 338 | 339 | 340 |
| 341 | 342 | 343 | 344 | 345 | 346 | 347 | 348 | 349 | 350 |
| 351 | 352 | 353 | 354 | 355 | 356 | 357 | 358 | 359 | 360 |
| 361 | 362 | 363 | 364 | 365 | 366 | 367 | 368 | 369 | 370 |
| 371 | 372 | 373 | 374 | 375 | 376 | 377 | 378 | 379 | 380 |
| 381 | 382 | 383 | 384 | 385 | 386 | 387 | 388 | 389 | 390 |
| 391 | 392 | 393 | 394 | 395 | 396 | 397 | 398 | 399 | 400 |
| 401 | 402 | 403 | 404 | 405 | 406 | 407 | 408 | 409 | 410 |
| 411 | 412 | 413 | 414 | 415 | 416 | 417 | 418 | 419 | 420 |
| 421 | 422 | 423 | 424 | 425 | 426 | 427 | 428 | 429 | 430 |
| 431 | 432 | 433 | 434 | 435 | 436 | 437 | 438 | 439 | 440 |
| 441 | 442 | 443 | 444 | 445 | 446 | 447 | 448 | 449 | 450 |
| 451 | 452 | 453 | 454 | 455 | 456 | 457 | 458 | 459 | 460 |
| 461 | 462 | 463 | 464 | 465 | 466 | 467 | 468 | 469 | 470 |
| 471 | 472 | 473 | 474 | 475 | 476 | 477 | 478 | 479 | 480 |
| 481 | 482 | 483 | 484 | 485 | 486 | 487 | 488 | 489 | 490 |
| 491 | 492 | 493 | 494 | 495 | 496 | 497 | 498 | 499 | 500 |
| 501 | 502 | 503 | 504 | 505 | 506 | 507 | 508 | 509 | 510 |
| 511 | 512 | 513 | 514 | 515 | 516 | 517 | 518 | 519 | 520 |
| 521 | 522 | 523 | 524 | 525 | 526 | 527 | 528 | 529 | 530 |
| 531 | 532 | 533 | 534 | 535 | 536 | 537 | 538 | 539 | 540 |
| 541 | 542 | 543 | 544 | 545 | 546 | 547 | 548 | 549 | 550 |
| 551 | 552 | 553 | 554 | 555 | 556 | 557 | 558 | 559 | 560 |
| 561 | 562 | 563 | 564 | 565 | 566 | 567 | 568 | 569 | 570 |
| 571 | 572 | 573 | 574 | 575 | 576 | 577 | 578 | 579 | 580 |
| 581 | 582 | 583 | 584 | 585 | 586 | 587 | 588 | 589 | 590 |
| 591 | 592 | 593 | 594 | 595 | 596 | 597 | 598 | 599 | 600 |
| 601 | 602 | 603 | 604 | 605 | 606 | 607 | 608 | 609 | 610 |
| 611 | 612 | 613 | 614 | 615 | 616 | 617 | 618 | 619 | 620 |
| 621 | 622 | 623 | 624 | 625 | 626 | 627 | 628 | 629 | 630 |
| 631 | 632 | 633 | 634 | 635 | 636 | 637 | 638 | 639 | 640 |
| 641 | 642 | 643 | 644 | 645 | 646 | 647 | 648 | 649 | 650 |
| 651 | 652 | 653 | 654 | 655 | 656 | 657 | 658 | 659 | 660 |
| 661 | 662 | 663 | 664 | 665 | 666 | 667 | 668 | 669 | 670 |
| 671 | 672 | 673 | 674 | 675 | 676 | 677 | 678 | 679 | 680 |
| 681 | 682 | 683 | 684 | 685 | 686 | 687 | 688 | 689 | 690 |
| 691 | 692 | 693 | 694 | 695 | 696 | 697 | 698 | 699 | 700 |
| 701 | 702 | 703 | 704 | 705 | 706 | 707 | 708 | 709 | 710 |
| 711 | 712 | 713 | 714 | 715 | 716 | 717 | 718 | 719 | 720 |
| 721 | 722 | 723 | 724 | 725 | 726 | 727 | 728 | 729 | 730 |
| 731 | 732 | 733 | 734 | 735 | 736 | 737 | 738 | 739 | 740 |
| 741 | 742 | 743 | 744 | 745 | 746 | 747 | 748 | 749 | 750 |
| 751 | 752 | 753 | 754 | 755 | 756 | 757 | 758 | 759 | 760 |
| 761 | 762 | 763 | 764 | 765 | 766 | 767 | 768 | 769 | 770 |
| 771 | 772 | 773 | 774 | 775 | 776 | 777 | 778 | 779 | 780 |
| 781 | 782 | 783 | 784 | 785 | 786 | 787 | 788 | 789 | 790 |
| 791 | 792 | 793 | 794 | 795 | 796 | 797 | 798 | 799 | 800 |
| 801 | 802 | 803 | 804 | 805 | 806 | 807 | 808 | 809 | 810 |
| 811 | 812 | 813 | 814 | 815 | 816 | 817 | 818 | 819 | 820 |
| 821 | 822 | 823 | 824 | 825 | 826 | 827 | 828 | 829 | 830 |
| 831 | 832 | 833 | 834 | 835 | 836 | 837 | 838 | 839 | 840 |
| 841 | 842 | 843 | 844 | 845 | 846 | 847 | 848 | 849 | 850 |
| 851 | 852 | 853 | 854 | 855 | 856 | 857 | 858 | 859 | 860 |
| 861 | 862 | 863 | 864 | 865 | 866 | 867 | 868 | 869 | 870 |
| 871 | 872 | 873 | 874 | 875 | 876 | 877 | 878 | 879 | 880 |
| 881 | 882 | 883 | 884 | 885 | 886 | 887 | 888 | 889 | 890 |
| 891 | 892 | 893 | 894 | 895 | 896 | 897 | 898 | 899 | 900 |
| 901 | 902 | 903 | 904 | 905 | 906 | 907 | 908 | 909 | 910 |
| 911 | 912 | 913 | 914 | 915 | 916 | 917 | 918 | 919 | 920 |
| 921 | 922 | 923 | 924 | 925 | 926 | 927 | 928 | 929 | 930 |
| 931 | 932 | 933 | 934 | 935 | 936 | 937 | 938 | 939 | 940 |
| 941 | 942 | 943 | 944 | 945 | 946 | 947 | 948 | 949 | 950 |
| 951 | 952 | 953 | 954 | 955 | 956 | 957 | 958 | 959 | 960 |
| 961 | 962 | 963 | 964 | 965 | 966 | 967 | 968 | 969 | 970 |
| 971 | 972 | 973 | 974 | 975 | 976 | 977 | 978 | 979 | 980 |
| 981 | 982 | 983 | 984 | 985 | 986 | 987 | 988 | 989 | 990 |
| 991 | 992 | 993 | 994 | 995 | 996 | 997 | 998 | 999 | 1000 |
| 1001 | 1002 | 1003 | 1004 | 1005 | 1006 | 1007 | 1008 | 1009 | 1010 |
| 1011 | 1012 | 1013 | 1014 | 1015 | 1016 | 1017 | 1018 | 1019 | 1020 |
| 1021 | 1022 | 1023 | 1024 | 1025 | 1026 | 1027 | 1028 | 1029 | 1030 |
| 1031 | 1032 | 1033 | 1034 | 1035 | 1036 | 1037 | 1038 | 1039 | 1040 |
| 1041 | 1042 | 1043 | 1044 | 1045 | 1046 | 1047 | 1048 | 1049 | 1050 |
| 1051 | 1052 | 1053 | 1054 | 1055 | 1056 | 1057 | 1058 | 1059 | 1060 |
| 1061 | 1062 | 1063 | 1064 | 1065 | 1066 | 1067 | 1068 | 1069 | 1070 |
| 1071 | 1072 | 1073 | 1074 | 1075 | 1076 | 1077 | 1078 | 1079 | 1080 |
| 1081 | 1082 | 1083 | 1084 | 1085 | 1086 | 1087 | 1088 | 1089 | 1090 |
| 1091 | 1092 | 1093 | 1094 | 1095 | 1096 | 1097 | 1098 | 1099 | 1100 |
| 1101 | 1102 | 1103 | 1104 | 1105 | 1106 | 1107 | 1108 | 1109 | 1110 |
| 1111 | 1112 | 1113 | 1114 | 1115 | 1116 | 1117 | 1118 | 1119 | 1120 |
| 1121 | 1122 | 1123 | 1124 | 1125 | 1126 | 1127 | 1128 | 1129 | 1130 |
| 1131 | 1132 | 1133 | 1134 | 1135 | 1136 | 1137 | 1138 | 1139 | 1140 |
| 1141 | 1142 | 1143 | 1144 | 1145 | 1146 | 1147 | 1148 | 1149 | 1150 |
| 1151 | 1152 | 1153 | 1154 | 1155 | 1156 | 1157 | 1158 | 1159 | 1160 |
| 1161 | 1162 | 1163 | 1164 | 1165 | 1166 | 1167 | 1168 | 1169 | 1170 |
| 1171 | 1172 | 1173 | 1174 | 1175 | 1176 | 1177 | 1178 | 1179 | 1180 |
| 1181 | 1182 | 1183 | 1184 | 1185 | 1186 | 1187 | 1188 | 1189 | 1190 |
| 1191 | 1192 | 1193 | 1194 | 1195 | 1196 | 1197 | 1198 | 1199 | 1200 |
| 1201 | 1202 | 1203 | 1204 | 1205 | 1206 | 1207 | 1208 | 1209 | 1210 |
| 1211 | 1212 | 1213 | 1214 | 1215 | 1216 | 1217 | 1218 | 1219</ | |

| File | | Keying Instruction | | Graphic | | Description | | Page of | |
|------------|------|--------------------|--|---------|--|-------------|--|---------|--|
| Programmer | Date | | | Key | | | | | |

| Sequence Number | Form Type | Conditioning | | | | Name | Reference ID | Length | Data Type (B/N/P/S/B/F/A/X/Y/I/N/D/W) | Decimal Positions | Usage (N/O/I/B/N/A/N/P) | Location | | Functions |
|-----------------|-----------|--------------|------------|------------|------------|----------|--------------|--------|---------------------------------------|-------------------|-------------------------|----------|-----|-----------------------------------|
| | | Indic. (0) | Indic. (1) | Indic. (2) | Indic. (3) | | | | | | | Line | Pos | |
| 1 | A | | | | | WHSLOC | | 3 | | | | 48 | 49 | CHECK(MF) |
| 2 | A | | | | | | | | | | | 50 | 51 | COLHDG('BIN' 'NO') |
| 3 | A | | | | | ORDDAT | | 6S | 0 | | | 52 | 53 | TEXT('DATA ORDER WAS ENTERED') |
| 4 | A | | | | | CUSTYP | | 1S | 0 | | | 54 | 55 | RANGE(1 5) |
| 5 | A | | | | | | | | | | | 56 | 57 | COLHDG('CUST' 'TYPE') |
| 6 | A | | | | | | | | | | | 58 | 59 | TEXT('CUSTOMER TYPE 1-GOV 2-SCH + |
| 7 | A | | | | | | | | | | | 60 | 61 | 3-BUS 4-PVT 5-OTH') |
| 8 | A | | | | | STATE | | 2 | | | | 62 | 63 | CHECK(MF) |
| 9 | A | | | | | | | | | | | 64 | 65 | COLHDG('STATE') |
| 10 | A | | | | | ACTMTH | | 2S | 0 | | | 66 | 67 | COLHDG('ACCT' 'MTH') |
| 11 | A | | | | | | | | | | | 68 | 69 | TEXT('ACCOUNTING MONTH OF SALE') |
| 12 | A | | | | | ACTYR | | 2S | 0 | | | 70 | 71 | COLHDG('ACCT' 'YEAR') |
| 13 | A | | | | | | | | | | | 72 | 73 | TEXT('ACCOUNTING YEAR OF SALE') |
| 14 | A | | | | | K ORDER | | | | | | | | |
| 15 | A | | | | | K LINNUM | | | | | | | | |

Figure 8-11 (Part 8 of 15). Program and Supporting DDS for Using Subfiles

USING DISPLAY FILES

```

5728PL1 R01 M00 880715          PL/I Source Listing          LP1422/LP1422          11/30/88 15:39:54          Page 2
LP1422: PROCEDURE;
*<.....1.....2.....3.....4.....5.....6.....7.....>.....8 Date
LP1422: PROCEDURE;
100      1
200
300
400      2      1 1          /* FILE DECLARATIONS */
500      1 1          DECLARE          1          2          3          4
600      2.1    1 1          ORD220D FILE RECORD SEQUENTIAL KEYED UPDATE ENV(INTERACTIVE),
700      2.2    1 1          ORDHDRP FILE RECORD DIRECT INPUT ENV(INDEXED DESCRIBED),
800      2.3    1 1          ORDDTLP FILE RECORD DIRECT INPUT ENV(INDEXED DESCRIBED),
900          CUSHSTP FILE RECORD DIRECT INPUT ENV(INDEXED DESCRIBED);
1000          4          5          6          7
1100     3      1 1          /* RECORD DECLARATIONS */
1200     1 1          DECLARE
1300     1 SUB1_RECORD,
SUB1      + 100          8          %INCLUDE ORD220D(SUB1,OUTPUT,,COMMA);
SUB1      + 200          /* ----- */
SUB1      + 300          /* DEVICE FILE: ORD220D.LP1422 */
SUB1      + 400          /* FILE CREATION DATE: 87/11/30 */
SUB1      + 500          /* RECORD FORMAT: SUB1 */
SUB1      + 600          /* RECORD FORMAT SEQUENCE ID: 14DC3D225CC47 */
SUB1      + 700     3.1    1 1          /* ----- */
SUB1      + 800     3.2    1 1          15 ITEM      PIC '9999R',      /* Item Number */
SUB1      + 900     3.3    1 1          15 QTYORD   PIC '99R',      /* Quantity Ordered */
SUB1      + 1000    3.4    1 1          15 DESCRP  CHAR(30),     /* Item Description */
SUB1      + 1100   3.5    1 1          15 PRICE   PIC '9999V9R', /* Selling Price */
SUB1      + 1200   3.5    1 1          15 EXTENS  PIC '999999V9R', /* Extension Amount of QTYORD */
SUB1      + 1400   3.6    1 1          PRICE
SUB1      + 1500   3.6    1 1          1 SUBCTL1_RECORD_INPUT,
SUBCTL1   + 100          9          %INCLUDE ORD220D(SUBCTL1,INPUT,,COMMA);
SUBCTL1   + 200          /* ----- */
SUBCTL1   + 300          /* DEVICE FILE: ORD220D.LP1422 */
SUBCTL1   + 400          /* FILE CREATION DATE: 87/11/30 */
SUBCTL1   + 500          /* RECORD FORMAT: SUBCTL1 */
SUBCTL1   + 600          /* RECORD FORMAT SEQUENCE ID: 1AC3AF315286B */
SUBCTL1   + 700          /* ----- */
SUBCTL1   + 800          /* INDICATORS FOR FORMAT SUBCTL1 */
SUBCTL1   + 900          /* INDICATOR 45 */
SUBCTL1   + 1000         /* INDICATOR 47      No lines for this order */
SUBCTL1   + 1100         /* INDICATOR 57      Display Subfile */
SUBCTL1   + 1200         /* INDICATOR 58      Off=Display Subctl On=Clear Subfile */
SUBCTL1   + 1300         /* INDICATOR 61      Order number not found */
SUBCTL1   + 1400         /* INDICATOR 62      No customer record found for this order */
SUBCTL1   + 1500         /* INDICATOR 97      Continue Display */
SUBCTL1   + 1600         /* INDICATOR 99      End of Program */
SUBCTL1   + 1700         /* ----- */
SUBCTL1   + 1800     3.7    1 1          15 ORDER    PIC '9999R',      /* Order Number */
SUBCTL1   + 1900     3.8    1 1          1 SUBCTL1_RECORD_OUTPUT,
SUBCTL1   + 200          10         %INCLUDE ORD220D(SUBCTL1,OUTPUT,,COMMA);
SUBCTL1   + 300          /* ----- */
SUBCTL1   + 400          /* DEVICE FILE: ORD220D.LP1422 */
SUBCTL1   + 500          /* FILE CREATION DATE: 87/11/30 */
SUBCTL1   + 600          /* RECORD FORMAT: SUBCTL1 */
SUBCTL1   + 700          /* RECORD FORMAT SEQUENCE ID: 1AC3AF315286B */
SUBCTL1   + 800          /* ----- */
SUBCTL1   + 900          /* INDICATORS FOR FORMAT SUBCTL1

```

Figure 8-11 (Part 9 of 15). Program and Supporting DDS for Using Subfiles

```

5728PL1 R01 M00 880715          PL/I Source Listing          LP1422/LP1422          11/30/88 15:39:54          Page 3
                                LP1422: PROCEDURE;          PRI00160
Include      SEQNBR STMT.SUBS BLK BN DO  *<..+...1...+...2...+...3...+...4...+...5...+...6...+...7...>+...8 Date
SUBCTL1 + 800          /* INDICATOR 45          /*
SUBCTL1 + 900          /* INDICATOR 47          No lines for this order /*
SUBCTL1 + 1000         /* INDICATOR 57          Display Subfile         /*
SUBCTL1 + 1100         /* INDICATOR 58          Off=Display Subctl On=Clear Subfile /*
SUBCTL1 + 1200         /* INDICATOR 61          Order number not found  /*
SUBCTL1 + 1300         /* INDICATOR 62          No customer record found for this order /*
SUBCTL1 + 1400         /* INDICATOR 97          Continue Display        /*
SUBCTL1 + 1500         /* INDICATOR 99          End of Program          /*
SUBCTL1 + 1600         /* ----- /*
SUBCTL1 + 1700         3.9      1 1          15 ORDER PIC '9999R', /* Order Number          /*
SUBCTL1 + 1800         3.10    1 1          15 ORDDAT PIC '99999R', /* Date Order was Entered /*
SUBCTL1 + 1900         3.11    1 1          15 CUST CHAR(5), /* Customer Number       /*
SUBCTL1 + 2000         3.12    1 1          15 NAME CHAR(25), /* Customer Name         /*
SUBCTL1 + 2100         3.13    1 1          15 ADDR CHAR(20), /* Customer Address      /*
SUBCTL1 + 2200         3.14    1 1          15 CITY CHAR(20), /* Customer City         /*
SUBCTL1 + 2300         3.15    1 1          15 STATE CHAR(2), /* State                 /*
SUBCTL1 + 2400         3.16    1 1          15 ZIP PIC '9999R', /* Zip Code              /*
SUBCTL1 + 2500         3.17    1 1          15 ORDAMT PIC '999999V9R', /* Total Dollar Amount of the /*
SUBCTL1 + 2600         /* Order /*
SUBCTL1 + 2700         3.18    1 1          15 STSORD CHAR(12), /*
SUBCTL1 + 2800         3.19    1 1          15 STSOPN CHAR(12), /*
SUBCTL1 + 2900         3.20    1 1          15 CUSORD CHAR(15), /* Customer Purchase Order Number /*
SUBCTL1 + 3000         /* /*
SUBCTL1 + 3100         3.21    1 1          15 SHPVIA CHAR(15), /* Shipping Instructions  /*
SUBCTL1 + 3200         3.22    1 1          15 PRTOAT PIC '99999R', /* Date Order was Printed /*
SUBCTL1 + 3300         3.23    1 1          15 INVNUM PIC '9999R', /* Invoice Number         /*
SUBCTL1 + 3400         3.24    1 1          15 ACTMTH PIC '9R', /* Accounting Month of Sale /*
SUBCTL1 + 3500         3.25    1 1          15 ACTYR PIC '9R', /* Accounting Year of Sale /*
SUBCTL1 + 1800         3.26    1 1          1 SUBCTL1_INDICATORS,
SUBCTL1 + 1900         /* %INCLUDE ORD220D(SUBCTL1,INDICATORS,,COMMA);
SUBCTL1 + 100         /* ----- /*
SUBCTL1 + 200         /* DEVICE FILE: ORD220D.LP1422 /*
SUBCTL1 + 300         /* FILE CREATION DATE: 87/11/30 /*
SUBCTL1 + 400         /* RECORD FORMAT: SUBCTL1 /*
SUBCTL1 + 500         /* RECORD FORMAT SEQUENCE ID: 1AC3AF315286B /*
SUBCTL1 + 600         /* ----- /*
SUBCTL1 + 700         3.27    1 1          15 IN01_IN44 CHAR(44), /* UNDEFINED INDICATOR(S) /*
SUBCTL1 + 800         3.28    1 1          15 IN45 PIC '9', /*
SUBCTL1 + 900         3.29    1 1          15 IN46_IN46 CHAR(01), /* UNDEFINED INDICATOR(S) /*
SUBCTL1 + 1000        3.30    1 1          15 IN47 PIC '9', /* No lines for this order /*
SUBCTL1 + 1100        3.31    1 1          15 IN48_IN56 CHAR(09), /* UNDEFINED INDICATOR(S) /*
SUBCTL1 + 1200        3.32    1 1          15 IN57 PIC '9', /* Display Subfile       /*
SUBCTL1 + 1300        3.33    1 1          15 IN58 PIC '9', /* Off=Display Subctl On=Clear /*
SUBCTL1 + 1400         /* Subfile /*
SUBCTL1 + 1500        3.34    1 1          15 IN59_IN60 CHAR(02), /* UNDEFINED INDICATOR(S) /*
SUBCTL1 + 1600        3.35    1 1          15 IN61 PIC '9', /* Order number not found /*
SUBCTL1 + 1700        3.36    1 1          15 IN62 PIC '9', /* No customer record found for /*
SUBCTL1 + 1800         /* this order /*
SUBCTL1 + 1900        3.37    1 1          15 IN63_IN96 CHAR(34), /* UNDEFINED INDICATOR(S) /*
SUBCTL1 + 2000        3.38    1 1          15 IN97 PIC '9', /* Continue Display      /*
SUBCTL1 + 2100        3.39    1 1          15 IN98_IN98 CHAR(01), /* UNDEFINED INDICATOR(S) /*
SUBCTL1 + 2200        3.40    1 1          15 IN99 PIC '9', /* End of Program        /*
SUBCTL1 + 2000        3.41    1 1          1 ORDER_HEADER,
                                831007
                                831007
                                PRI00290

```

Figure 8-11 (Part 10 of 15). Program and Supporting DDS for Using Subfiles

USING DISPLAY FILES

```

5728PL1 R01 M00 880715          PL/I Source Listing          LP1422/LP1422          11/30/88 15:39:54          Page 4
                                LP1422: PROCEDURE;
                                *<.....1.....2.....3.....4.....5.....6.....7.....>.....8 Date
Include      SEQNBR STMT.SUBS BLK BN DO  %INCLUDE ORDHDRP(ORDHDR,RECORD,,COMMA);          PRI00160
                                %INCLUDE ORDHDRP(ORDHDR,RECORD,,COMMA);          PRI00300 830919
ORDHDR      + 100          /* ----- */
ORDHDR      + 200          /* PHYSICAL FILE: ORDHDRP.LP1422          */
ORDHDR      + 300          /* FILE CREATION DATE: 87/11/30          */
ORDHDR      + 400          /* RECORD FORMAT: ORDHDR          */
ORDHDR      + 500          /* RECORD FORMAT SEQUENCE ID: 4F2FC74D337EB          */
ORDHDR      + 600          /* -----Order Header Record----- */
ORDHDR      + 700      3.42      1 1          15 CUST      CHAR(5),          /* Customer Number          */
ORDHDR      + 800      3.43      1 1          15 ORDER      PIC '9999R',          /* Order Number          */
ORDHDR      + 900          /* DDS - KEY FIELD          */
ORDHDR      + 1000     3.44      1 1          15 ORDDAT     PIC '99999R',          /* Date Order was Entered          */
ORDHDR      + 1100     3.45      1 1          15 CUSORD     CHAR(15),          /* Customer Purchase Order Number          */
ORDHDR      + 1200          /* ----- */
ORDHDR      + 1300     3.46      1 1          15 SHPVIA     CHAR(15),          /* Shipping Instructions          */
ORDHDR      + 1400     3.47      1 1          15 ORDSTS     PIC 'R',          /* Order Status 1=PCS 2=CNT 3=CHK          */
ORDHDR      + 1500          /* 4=RDY 5=PRT 6=PCK          */
ORDHDR      + 1600     3.48      1 1          15 OPRNAM     CHAR(10),          /* Operator Name who entered the          */
ORDHDR      + 1700          /* order          */
ORDHDR      + 1800     3.49      1 1          15 ORDAMT     PIC '999999V9R',          /* Total Dollar Amount of the          */
ORDHDR      + 1900          /* Order          */
ORDHDR      + 2000     3.50      1 1          15 CUSTYP     PIC 'R',          /* Customer Type 1=GOV 2=SCH          */
ORDHDR      + 2100          /* 3=BUS 4=PVT 5=OTH          */
ORDHDR      + 2200     3.51      1 1          15 INVNUM     PIC '9999R',          /* Invoice Number          */
ORDHDR      + 2300     3.52      1 1          15 PRDAT      PIC '99999R',          /* Date Order was Printed          */
ORDHDR      + 2400     3.53      1 1          15 OPHSTS     PIC 'R',          /* Order Open Status 1=OPEN          */
ORDHDR      + 2500          /* 2=CLOSE 3=CANCEL          */
ORDHDR      + 2600     3.54      1 1          15 TOTLIN     PIC '99R',          /* Total Line Items in Order          */
ORDHDR      + 2700     3.55      1 1          15 ACTMTH     PIC '9R',          /* Accounting Month of Sale          */
ORDHDR      + 2800     3.56      1 1          15 ACTYR     PIC '9R',          /* Accounting Year of Sale          */
ORDHDR      + 2900     3.57      1 1          15 STATE     CHAR(2),          /* State          */
ORDHDR      + 3000     3.58      1 1          15 AMPAID     PIC '999999V9R',          /* Total Dollar Amount Paid          */
                                1 ORDER_DETAIL,          PRI00310
                                %INCLUDE ORDDTLP(ORDDTL,RECORD,,COMMA);          PRI00320 830919
ORDDTL      + 100          /* ----- */
ORDDTL      + 200          /* PHYSICAL FILE: ORDDTLP.LP1422          */
ORDDTL      + 300          /* FILE CREATION DATE: 87/11/30          */
ORDDTL      + 400          /* RECORD FORMAT: ORDDTL          */
ORDDTL      + 500          /* RECORD FORMAT SEQUENCE ID: 3CA0F801BF74E          */
ORDDTL      + 600          /* -----Order Detail Record----- */
ORDDTL      + 700      3.60      1 1          15 CUST      CHAR(5),          /* Customer Number          */
ORDDTL      + 800      3.61      1 1          15 ORDER      PIC '9999R',          /* Order Number          */
ORDDTL      + 900          /* DDS - KEY FIELD          */
ORDDTL      + 1000     3.62      1 1          15 LINNUM     PIC '99R',          /* Line Number of line in order*/
ORDDTL      + 1100          /* DDS - KEY FIELD          */
ORDDTL      + 1200     3.63      1 1          15 ITEM      PIC '9999R',          /* Item Number          */
ORDDTL      + 1300     3.64      1 1          15 QTYORD     PIC '99R',          /* Quantity Ordered          */
ORDDTL      + 1400     3.65      1 1          15 DESCRP     CHAR(30),          /* Item Description          */
ORDDTL      + 1500     3.66      1 1          15 PRICE     PIC '99999V9R',          /* Selling Price          */
ORDDTL      + 1600     3.67      1 1          15 EXTENS     PIC '999999V9R',          /* Extension Amount of QTYORD *          */
ORDDTL      + 1700          /* PRICE          */
ORDDTL      + 1800     3.68      1 1          15 WHSLOC     CHAR(3),          /* Bin No.          */
ORDDTL      + 1900     3.69      1 1          15 ORDDAT     PIC '99999R',          /* Date Order was Entered          */
ORDDTL      + 2000     3.70      1 1          15 CUSTYP     PIC 'R',          /* Customer Type 1=GOV 2=SCH

```

Figure 8-11 (Part 11 of 15). Program and Supporting DDS for Using Subfiles


```

5728PL1 R01 M00 888715          PL/I Source Listing          LP1422/LP1422          11/30/88 15:39:54          Page 5
                                LP1422: PROCEDURE;          PRI00160
Include  SEQNBR STMT.SUBS BLK BN DO  *<.....1.....2.....3.....4.....5.....6.....7.....8 Date
ORDDTL  + 2100                                3=BUS 4=PVT 5=OTH          */
ORDDTL  + 2200 3.71 1 1          15 STATE CHAR(2),          /* State          */
ORDDTL  + 2300 3.72 1 1          15 ACTMTH PIC '9R',          /* Accounting Month of Sale */
ORDDTL  + 2400 3.73 1 1          15 ACTYR PIC '9R',          /* Accounting Year of Sale */
        2400 3.74 1 1          1 CUSTOMER_MASTER,
        2500                                %INCLUDE_CUSMSTP(CUSMST,RECORD);          830603
CUSMST  + 100                                /* ----- */
CUSMST  + 200                                /* PHYSICAL FILE: CUSMSTP.LP1422          */
CUSMST  + 300                                /* FILE CREATION DATE: 87/11/30          */
CUSMST  + 400                                /* RECORD FORMAT: CUSMST          */
CUSMST  + 500                                /* RECORD FORMAT SEQUENCE ID: 4AA7C90E9BB1E          */
CUSMST  + 600                                /* -----CUSTOMER MASTER RECORD----- */
CUSMST  + 700 3.75 1 1          15 CUST CHAR(5),          /* CUSTOMER NUMBER          */
CUSMST  + 800                                /* DDS - KEY FIELD          */
CUSMST  + 900 3.76 1 1          15 NAME CHAR(25),          /* CUSTOMER NAME          */
CUSMST  + 1000 3.77 1 1          15 ADDR CHAR(20),          /* CUSTOMER ADDRESS          */
CUSMST  + 1100 3.78 1 1          15 CITY CHAR(20),          /* CUSTOMER CITY          */
CUSMST  + 1200 3.79 1 1          15 STATE CHAR(2),          /* STATE          */
CUSMST  + 1300 3.80 1 1          15 ZIP PIC '9999R',          /* ZIP CODE          */
CUSMST  + 1400 3.81 1 1          15 SRHCOD CHAR(6),          /* CUSTOMER NUMBER SEARCH CODE */
CUSMST  + 1500 3.82 1 1          15 CUSTYP PIC 'R',          /* CUSTOMER TYPE 1=60V 2=SCH          */
CUSMST  + 1600                                3=BUS 4=PVT 5=OT          */
CUSMST  + 1700 3.83 1 1          15 ARBAL PIC '999999V9R', /* ACCTS REC BALANCE          */
CUSMST  + 1800 3.84 1 1          15 ORDBAL PIC '999999V9R', /* A/R AMT IN ORDER FILE          */
CUSMST  + 1900 3.85 1 1          15 LSTAMT PIC '999999V9R', /* LAST AMT PAID IN A/R          */
CUSMST  + 2000 3.86 1 1          15 LSTDAT PIC '99999R', /* LAST DATE PAID IN A/R          */
CUSMST  + 2100 3.87 1 1          15 CRDLMT PIC '999999V9R', /* CUSTOMER CREDIT LIMIT          */
CUSMST  + 2200 3.88 1 1          15 SLSYR PIC '99999999V9R',
CUSMST  + 2300                                /* CUSTOMER SALES THIS YEAR          */
CUSMST  + 2400 3.89 1 1          15 SLSLYR PIC '99999999V9R';
CUSMST  + 2500                                /* CUSTOMER SALES LAST YEAR          */
        2600                                830603
        2700                                /* KEY STRUCTURE */          830603
        2800 4 1 1          DECLARE          830603
        2900 1 1          1 DETAIL_KEY,          830602
        3000 4.1 1 1          2 D_ORDER PICTURE '9999R',          831011
        3100 4.2 1 1          2 D_LINNUM PICTURE '99R';          831011
        3200                                PRI00330
        3300                                /* INDICATOR FLAGS */          PRI00340
        3400 5 1 1          DECLARE          PRI00350
        3500 1 1          1 INDICATOR_FLAGS STATIC,          PRI00360
        3600 5.1 1 1          2 OFF PICTURE '9' INIT(0),          PRI00370 831007
        3700 5.2 1 1          2 ON PICTURE '9' INIT(1);          PRI00380 831007
        3800                                PRI00390
        3900                                /* BUILT-IN FUNCTIONS */          PRI00470 830602
        4000 6 1 1          DECLARE          PRI00480
        4100 1 1          ONCODE BUILTIN,          PRI00490 831011
        4200 6.1 1 1          DECIMAL BUILTIN;          831011
        4300                                830602
        4400                                /* PROGRAM VARIABLES */          PRI00580
        4500 7 1 1          DECLARE          PRI00590
        4600 1 1          OPNSTAT(3) STATIC CHAR(12) INIT('1-OPEN',          PRI00600 830609
        4700 1 1          '2-CLOSED',          830609
    
```

Figure 8-11 (Part 12 of 15). Program and Supporting DDS for Using Subfiles

USING DISPLAY FILES

| 5728PL1 R01 M00 880715 | PL/I Source Listing | LP1422/LP1422 | 11/30/88 15:39:54 | Page 6 | |
|------------------------|----------------------------|--------------------|--|--------------------------------------|-----------------|
| Include | SEQNBR STMT.SUBS BLK BN DO | LP1422: PROCEDURE; | *<..+....1....+....2....+....3....+....4....+....5....+....6....+....7.>..+....8 | Date | |
| | 4800 | 1 1 | | '3-CANCELLED'), | 830609 |
| | 4900 | 7.1 1 1 | ORDSTAT(9) | STATIC CHAR(12) INIT('1-IN PROCESS', | 830609 |
| | 5000 | 1 1 | | '2-CONTINUED', | 830609 |
| | 5100 | 1 1 | | '3-CREDIT CHK', | 830609 |
| | 5200 | 1 1 | | '4-READY PRT', | 830609 |
| | 5300 | 1 1 | | '5-PRINTED', | 830609 |
| | 5400 | 1 1 | | '6-PICKED', | 830609 |
| | 5500 | 1 1 | | '7-INVOICED', | 830609 |
| | 5600 | 1 1 | | '8-INVALID', | 830609 |
| | 5700 | 1 1 | | '9-CANCELLED'), | 830609 |
| | 5800 | 7.2 1 1 | READ_CHK | CHAR(5); | 830602 |
| | 5900 | | | | 830602 |
| | 6000 | 8 1 1 | 12 ON KEY(ORDHDRP) | | 830602 |
| | 6100 | 1 1 | BEGIN; | | 830602 |
| | 6200 | 9 2 2 | ON ERROR SYSTEM; | | 830602 |
| | 6300 | 10 2 2 | IF ONCODE = 51 THEN | | 830602 |
| | 6400 | 2 2 | 13 DO; | | 830602 |
| | 6500 | 11 2 2 1 | IN61 = ON; | | 830602 |
| | 6600 | 12 2 2 1 | GOTO DSP; | | 830602 |
| | 6700 | 13 2 2 1 | END; /* DO */ | | 830602 |
| | 6800 | 14 2 2 | END; /* BEGIN */ | | 830602 |
| | 6900 | | | | 830602 |
| | 7000 | 15 1 1 | ON KEY(ORDDTLP) | | 830602 |
| | 7100 | 1 1 | BEGIN; | | 830602 |
| | 7200 | 16 3 2 | ON ERROR SYSTEM; | | 830602 |
| | 7300 | 17 3 2 | IF ONCODE = 51 THEN | | 830602 |
| | 7400 | 3 2 | IF READ_CHK = 'READ1' THEN | | 830602 |
| | 7500 | 3 2 | DO; | | 830602 |
| | 7600 | 18 3 2 1 | IN47 = ON; | | 830602 |
| | 7700 | 19 3 2 1 | GOTO DSP; | | 830602 |
| | 7800 | 20 3 2 1 | END; /* DO */ | | 830602 |
| | 7900 | 21 3 2 | ELSE | | 830602 |
| | 8000 | 3 2 | GOTO SUB; | | 830602 |
| | 8100 | 22 3 2 | END; /* BEGIN */ | | 830602 |
| | 8200 | | | | 830602 |
| | 8300 | | | | 830602 |
| | 8400 | 23 1 1 | ON KEY(CUSMSTP) | | 830602 |
| | 8500 | 1 1 | BEGIN; | | 830602 |
| | 8600 | 24 4 2 | ON ERROR SYSTEM; | | 830602 |
| | 8700 | 25 4 2 | IF ONCODE = 51 THEN | | 830602 |
| | 8800 | 4 2 | DO; | | 830602 |
| | 8900 | 26 4 2 1 | IN62 = ON; | | 830602 |
| | 9000 | 27 4 2 1 | GOTO DSP; | | 830602 |
| | 9100 | 28 4 2 1 | END; /* DO */ | | 830602 |
| | 9200 | 29 4 2 | END; /* BEGIN */ | | 830602 |
| | 9300 | | | | 830610 |
| | 9400 | 30 1 1 | ON ENDFILE(ORD220D) | | PRI00620 830602 |
| | 9500 | 1 1 | GOTO SUB; | | PRI00630 830602 |
| | 9600 | | | | 830610 |
| | 9700 | 31 1 1 | ON ENDFILE(ORDDTLP) | | 830610 |
| | 9800 | 1 1 | GOTO SUB; | | 830610 |
| | 9900 | | | | PRI00640 |
| 10000 | | | /* MAIN PROGRAM */ | | PRI00650 |

Figure 8-11 (Part 13 of 15). Program and Supporting DDS for Using Subfiles

| 5728PL1 R01 M00 880715 | | PL/I Source Listing | | LP1422/LP1422 | 11/30/87 15:39:54 | Page 7 |
|------------------------|--------|---------------------|-----------|--|--|-----------------|
| Include | SEQNBR | STMT.SUBS | BLK BN DO | LP1422: PROCEDURE; | *<.....1.....2.....3.....4.....5.....6.....7.>.....8 | Date |
| | 10100 | 32 | 1 1 | SUBCTL1_INDICATORS = OFF; /* SET ALL INDICATORS OFF */ | | 880415 |
| | 10200 | | | | | 880415 |
| | 10300 | 33 | 1 1 | SUBCTL1_RECORD_OUTPUT.ORDDAT = 0; | | 880415 |
| | 10400 | 34 | 1 1 | SUBCTL1_RECORD_OUTPUT.ORDAMT = 0; | | 880415 |
| | 10500 | 35 | 1 1 | SUBCTL1_RECORD_OUTPUT.PRDTAT = 0; | | 880415 |
| | 10600 | 36 | 1 1 | SUBCTL1_RECORD_OUTPUT.ZIP = 0; | | 880415 |
| | 10700 | 37 | 1 1 | SUBCTL1_RECORD_OUTPUT.INVNUM = 0; | | 880415 |
| | 10800 | 38 | 1 1 | SUBCTL1_RECORD_OUTPUT.ACTMTH = 0; | | 880415 |
| | 10900 | 39 | 1 1 | SUBCTL1_RECORD_OUTPUT.ACTYR = 0; | | 880415 |
| | 11000 | | | | | 880415 |
| | 11100 | 40 | 1 1 | 14 OPEN | | PRI00690 880415 |
| | 11200 | | 1 1 | FILE (ORD220D); /* UPDATE */ | | PRI00700 880415 |
| | 11300 | 41 | 1 1 | 14 OPEN | | PRI00710 880415 |
| | 11400 | | 1 1 | FILE (ORDHDRP); /* INPUT */ | | PRI00720 880415 |
| | 11500 | 42 | 1 1 | 14 OPEN | | 880415 |
| | 11600 | | 1 1 | FILE (ORDDTLP); /* INPUT */ | | 880415 |
| | 11700 | 43 | 1 1 | 14 OPEN | | 880415 |
| | 11800 | | 1 1 | FILE (CUSMSTP); /* INPUT */ | | 880415 |
| | 11900 | | | | | PRI00750 880415 |
| | 12000 | 44 | 1 1 | 15 DSP: DO WHILE(IN99 = OFF); | | PRI00760 880415 |
| | 12100 | | | | | PRI00770 880415 |
| | 12200 | 45 | 1 1 1 | 16 WRITE FILE (ORD220D) FROM (SUBCTL1_RECORD_OUTPUT) | | PRI00780 880415 |
| | 12300 | | 1 1 1 | OPTIONS(RECORD('SUBCTL1') INDICATORS(SUBCTL1_INDICATORS)); | | PRI00790 880415 |
| | 12400 | 46 | 1 1 1 | IN58 = OFF; | | 880415 |
| | 12500 | 47 | 1 1 1 | 17 READ FILE (ORD220D) INTO (SUBCTL1_RECORD_INPUT) | | PRI00800 880415 |
| | 12600 | | 1 1 1 | OPTIONS(RECORD('SUBCTL1') INDICATORS(SUBCTL1_INDICATORS)); | | PRI00810 880415 |
| | 12700 | 48 | 1 1 1 | 18 READ FILE (ORDHDRP) INTO (ORDER_HEADER) | | PRI00820 880415 |
| | 12800 | | 1 1 1 | KEY(SUBCTL1_RECORD_INPUT.ORDER) OPTIONS(KEYSEARCH(EQUAL)); | | 880415 |
| | 12900 | 49 | 1 1 1 | 18 SUBCTL1_RECORD_OUTPUT = ORDER_HEADER, BY NAME; | | 880415 |
| | 13000 | 50 | 1 1 1 | 19 D_ORDER = SUBCTL1_RECORD_OUTPUT.ORDER; | | PRI00830 880415 |
| | 13100 | 51 | 1 1 1 | D_LINNUM = 1; | | PRI00840 880415 |
| | 13200 | 52 | 1 1 1 | READ_CHK = 'READ1'; | | PRI00850 880415 |
| | 13300 | 53 | 1 1 1 | 20 READ FILE (ORDDTLP) INTO (ORDER_DETAIL) KEY(DETAIL_KEY) | | PRI00860 880415 |
| | 13400 | | 1 1 1 | OPTIONS(KEYSEARCH(EQUAL) NBRKEYFLDS(2)); | | 880415 |
| | 13500 | 54 | 1 1 1 | 21 READ FILE (CUSMSTP) INTO (CUSTOMER_MASTER) | | PRI00870 880415 |
| | 13600 | | 1 1 1 | KEY(ORDER_DETAIL.CUST) OPTIONS(KEYSEARCH(EQUAL)); | | 880415 |
| | 13700 | 55 | 1 1 1 | 22 SUBCTL1_RECORD_OUTPUT = CUSTOMER_MASTER, BY NAME; | | 880415 |
| | 13800 | | | | | PRI00880 880415 |
| | 13900 | | | /* FILL THE SCREEN */ | | PRI00890 880415 |
| | 14000 | 56 | 1 1 1 | 23 DO WHILE(D_LINNUM < TOTLIN); | | PRI00910 880415 |
| | 14100 | 57 | 1 1 2 | SUB1_RECORD = ORDER_DETAIL, BY NAME; | | 880415 |
| | 14200 | 58 | 1 1 2 | 24 WRITE FILE (ORD220D) FROM (SUB1_RECORD) | | PRI00940 880415 |
| | 14300 | | 1 1 2 | KEYFROM(D_LINNUM) OPTIONS(RECORD('SUB1')); | | PRI00950 880415 |
| | 14400 | 59 | 1 1 2 | D_LINNUM = D_LINNUM + 1; | | 880415 |
| | 14500 | 60 | 1 1 2 | READ_CHK = 'READ2'; | | 880415 |
| | 14600 | 61 | 1 1 2 | READ FILE (ORDDTLP) INTO (ORDER_DETAIL) KEY(DETAIL_KEY) | | 880415 |
| | 14700 | | 1 1 2 | OPTIONS(KEYSEARCH(EQUAL)); | | 880415 |
| | 14800 | 62 | 1 1 2 | END; /* DO WHILE */ | | 880415 |
| | 14900 | | | | | 880415 |
| | 15000 | 63 | 1 1 1 | SUB1_RECORD = ORDER_DETAIL, BY NAME; | | 880415 |
| | 15100 | 64 | 1 1 1 | 24 WRITE FILE (ORD220D) FROM (SUB1_RECORD) | | PRI00940 880415 |
| | 15200 | | 1 1 1 | KEYFROM(D_LINNUM) OPTIONS(RECORD('SUB1')); | | PRI00950 880415 |
| | 15300 | 65 | 1 1 1 | SUB: IN57 = ON; | | 880415 |
| | 15400 | 66 | 1 1 1 | STSOPN = OPNSTAT(DECIMAL(OPNSTS)); | | 880415 |
| | 15500 | 67 | 1 1 1 | STSORD = ORDSTAT(DECIMAL(ORDSTS)); | | 880415 |
| | 15600 | 68 | 1 1 1 | 25 WRITE FILE (ORD220D) FROM (SUBCTL1_RECORD_OUTPUT) | | 880415 |
| | 15700 | | 1 1 1 | OPTIONS(RECORD('SUBCTL1') INDICATORS(SUBCTL1_INDICATORS)); | | 880415 |

Figure 8-11 (Part 14 of 15). Program and Supporting DDS for Using Subfiles

USING DISPLAY FILES

```

5728PL1 R01 M00 880715          PL/I Source Listing          LP1422/LP1422          11/30/88 15:39:54          Page 8
                                LP1422: PROCEDURE;          PRI00160
Include  SEQNBR STMT.SUBS BLK BN DO  *<.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.>.....8 Date
15800    69      1 1 1              IN57 = OFF;          880415
15900    70      1 1 1              IN58 = ON;          880415
16000    71      1 1 1              END; /* DO WHILE */  PRI00970 880415
16100
16200    72      1 1              CLOSE                PRI00980 880415
16300    1 1              FILE (ORD220D);      PRI00990 880415
16400    73      1 1              CLOSE                PRI01000 880415
16500    1 1              FILE (ORDHDRP);      PRI01010 880415
16600    74      1 1              CLOSE                880415
16700    1 1              FILE (ORDDTLP);      880415
16800    75      1 1              CLOSE                880415
16900    1 1              FILE (CUSMSTP);      880415
17000
17100    76      1 1              END LP1422;          PRI01020 880415
                                PRI01030 880415

```

Figure 8-11 (Part 15 of 15). Program and Supporting DDS for Using Subfiles

```

EXISTING ORDER INQUIRY          TOTAL $22,606.00
                                STATUS 4-READY PRT
                                OPEN 2-CLOSED
ORDER 12345 EVANS, NICK          CUSTOMER ORDER A375836743
DATE 88/06/14 123 LARKSONG AVE. SHIP VIA RUSH
CUST # 11111 NORTH YORK          PRINTED DATE 88/06/17
                                ON 98238
                                INVOICE 01401 MTH 06 YEAR 88

ITEM QTY ITEM DESCRIPTION          PRICE EXTENSION
11114 45 BITS AND BYTES          500.00 30.00
22228 10 ODDS AND ENDS          10.00 1.00
33332 6 LOTS OF STUFF          1.00 .60

```

Figure 8-12. Display Produced Using Supporting DDS for Subfiles

- 1 RECORD data transmission is used with ORD220D, ORDHDRP, ORDDTLP, and CUSMSTP.
- 2 ORD220D is declared as SEQUENTIAL KEYED, so that its records can be processed either sequentially or randomly by key. One of its fields, SUBCTL_RECORD_INPUT.ORDER, is used as a key when other files are accessed.
- 3 The UPDATE attribute and the INTERACTIVE option of the ENVIRONMENT attribute are specified for ORD220D. Both READ and WRITE statements are allowed, in this case to provide prompt screens and read input which is entered on the work station screen.

- 4** ORDHDRP, ORDDTLP, and CUSMSTP are declared with the **DIRECT** attribute, because they are accessed non-sequentially.
- 5** ORDHDRP, ORDDTLP, and CUSMSTP are used for **INPUT** only; no output is directed to the files by the program.
- 6** ORDHDRP, ORDDTLP, and CUSMSTP are **INDEXED**; the files are processed using the keyed sequence access path.
- 7** **DESCRIBED** indicates that external record format definitions are used in the program.
- 8** **SUB1_RECORD** consists of all the **OUTPUT** fields in record format **SUB1** of file **ORD220D**.
- 9** **SUBCTL1_RECORD_INPUT** consists of all the **INPUT** fields in record format **SUBCTL1** of file **ORD220D**.
- 10** **SUBCTL1_RECORD_OUTPUT** consists of all the **OUTPUT** fields in record format **SUBCTL1** of file **ORD220D**.
- 11** **SUBCTL1_INDICATORS** consists of all the **INDICATORS** in record format **SUBCTL1** of file **ORD220D**.
- 12** This on-unit runs when a **KEY** condition is raised for file **ORDHDRP**.
- 13** This do-group runs if the **KEY** condition is raised with code 51, that is, if no record in **ORDHDRP** has the key that is specified in a data transmission statement accessing the file.
- 14** The files are opened. The data transmission modes were specified as attributes in the file declaration; they could also have been specified here as options of the **OPEN** statement, but instead have been incorporated as comments.
- 15** Running of the do-loop continues while indicator 99 is set to **OFF**. When the user presses **F1**, indicator 99 is set to **ON**, running of the do-loop ends, the files are closed, and program running ends.
- 16** The prompt screen record is written to the display file. Record format **SUBCTL1** is used. **SUBCTL1_INDICATORS**, a structure defined earlier using a **%INCLUDE** directive, is passed to the display device.
- 17** The input supplied by the user is read into **SUBCTL1_RECORD_INPUT** using record format **SUBCTL1**, and the device returns its indicators to **SUBCTL1_INDICATORS**.
- 18** The key supplied by the user, field **ORDER** of structure **SUBCTL1_RECORD_INPUT**, is used to find and read a record from file **ORDHDRP** into **ORDER_HEADER**. The **OPTIONS** option of the **READ** statement specifies that the target key must be equal to the specified key.

The appropriate fields of **ORDER_HEADER** are then placed in **SUBCTL1_RECORD_OUTPUT** using a **BY NAME** assignment statement.

USING DISPLAY FILES

- 19** `DETAIL_KEY` is a structure consisting of the fields `D_ORDER` and `D_LINNUM`. The key supplied by the user, field `ORDER` of structure `SUBCTL1_RECORD_OUTPUT`, is assigned to `D_ORDER`. `D_LINNUM` is assigned an initial value of 1.
- 20** `DETAIL_KEY` is used to find and read into `ORDER_DETAIL` the first record in `ORDDTLP` with the order number specified by the user.
- 21** The customer number from the order detail record just read in is used as the key to read into `CUSTOMER_MASTER` the record from `CUSMSTP` containing information on the customer that placed the order.
- 22** The fields from `CUSTOMER_MASTER` that will be displayed on the work station screen are assigned to `SUBCTL1_RECORD_OUTPUT`.
- 23** The program enters a loop which reads in data using the key supplied by the user. A comparison with `TOTLIN` ensures that the loop ends when all records for that particular order have been read in from file `ORDDTLP`.
- 24** An order line is written to the subfile.
- 25** The new subfile is written to the device file `ORD220D`, and the order display appears on the work station screen.

Example of Using Indicators

Using Externally Defined Indicators in a Separate Area: The following program uses indicators in a separate area. They are included in the program with the %INCLUDE directive.

| File | | Keying Instruction | Graphic | | | | Description | Page of |
|------------|------|--------------------|---------|--|--|--|-------------|---------|
| Programmer | Date | | Key | | | | | |

| Sequence Number | Form Type | Conditioning | | | | Name | Length | Reference ID | Data Type | Position | Location | | Function |
|-----------------|-----------|--------------|-----------|-----------|-----------|------|--------|--------------|-----------|----------|----------|-----|----------|
| | | Indicator | Indicator | Indicator | Indicator | | | | | | Line | Pos | |
| 1 | A | | | | | | | | | | | | |
| 2 | A | | | | | | | | | | | | |
| 3 | A | | | | | | | | | | | | |
| 4 | A | | | | | | | | | | | | |
| 5 | A | | | | | | | | | | | | |
| 6 | A | | | | | | | | | | | | |
| 7 | A | | | | | | | | | | | | |
| 8 | A | | | | | | | | | | | | |
| 9 | A | | | | | | | | | | | | |
| 10 | A | | | | | | | | | | | | |
| 11 | A | | | | | | | | | | | | |
| 12 | A | | | | | | | | | | | | |
| 13 | A | | | | | | | | | | | | |
| 14 | A | | | | | | | | | | | | |
| 15 | A | | | | | | | | | | | | |
| 16 | A | | | | | | | | | | | | |
| 17 | A | | | | | | | | | | | | |
| 18 | A | | | | | | | | | | | | |
| 19 | A | | | | | | | | | | | | |
| 20 | A | | | | | | | | | | | | |
| 21 | A | | | | | | | | | | | | |
| 22 | A | | | | | | | | | | | | |
| 23 | A | | | | | | | | | | | | |
| 24 | A | | | | | | | | | | | | |
| 25 | A | | | | | | | | | | | | |
| 26 | A | | | | | | | | | | | | |
| 27 | A | | | | | | | | | | | | |
| 28 | A | | | | | | | | | | | | |
| 29 | A | | | | | | | | | | | | |
| 30 | A | | | | | | | | | | | | |
| 31 | A | | | | | | | | | | | | |
| 32 | A | | | | | | | | | | | | |
| 33 | A | | | | | | | | | | | | |
| 34 | A | | | | | | | | | | | | |
| 35 | A | | | | | | | | | | | | |
| 36 | A | | | | | | | | | | | | |
| 37 | A | | | | | | | | | | | | |
| 38 | A | | | | | | | | | | | | |
| 39 | A | | | | | | | | | | | | |
| 40 | A | | | | | | | | | | | | |
| 41 | A | | | | | | | | | | | | |
| 42 | A | | | | | | | | | | | | |
| 43 | A | | | | | | | | | | | | |
| 44 | A | | | | | | | | | | | | |
| 45 | A | | | | | | | | | | | | |
| 46 | A | | | | | | | | | | | | |
| 47 | A | | | | | | | | | | | | |
| 48 | A | | | | | | | | | | | | |
| 49 | A | | | | | | | | | | | | |
| 50 | A | | | | | | | | | | | | |
| 51 | A | | | | | | | | | | | | |
| 52 | A | | | | | | | | | | | | |
| 53 | A | | | | | | | | | | | | |
| 54 | A | | | | | | | | | | | | |
| 55 | A | | | | | | | | | | | | |
| 56 | A | | | | | | | | | | | | |
| 57 | A | | | | | | | | | | | | |
| 58 | A | | | | | | | | | | | | |
| 59 | A | | | | | | | | | | | | |
| 60 | A | | | | | | | | | | | | |
| 61 | A | | | | | | | | | | | | |
| 62 | A | | | | | | | | | | | | |
| 63 | A | | | | | | | | | | | | |
| 64 | A | | | | | | | | | | | | |
| 65 | A | | | | | | | | | | | | |
| 66 | A | | | | | | | | | | | | |
| 67 | A | | | | | | | | | | | | |
| 68 | A | | | | | | | | | | | | |
| 69 | A | | | | | | | | | | | | |
| 70 | A | | | | | | | | | | | | |
| 71 | A | | | | | | | | | | | | |
| 72 | A | | | | | | | | | | | | |
| 73 | A | | | | | | | | | | | | |
| 74 | A | | | | | | | | | | | | |
| 75 | A | | | | | | | | | | | | |
| 76 | A | | | | | | | | | | | | |
| 77 | A | | | | | | | | | | | | |
| 78 | A | | | | | | | | | | | | |
| 79 | A | | | | | | | | | | | | |
| 80 | A | | | | | | | | | | | | |

Figure 8-13 (Part 1 of 3). Program Using Indicators in a Separate Area

USING DISPLAY FILES

| 5728PL1 R01 M00 880715 | | PL/I Source Listing | | LP1423/LP1423 | | 11/30/88 15:30:01 | | Page 2 | | |
|------------------------|--------|---------------------|-----------|---|--|----------------------|---------------------------|-------------------|-----------------|----------|
| Include | SEQNBR | STMT.SUBS | BLK BN DO | *<.....1.....2.....3.....4.....5.....6.....7.>.....8 Date | | | | SEP00160 | | |
| | 100 | 1 | | LP1423: PROCEDURE; | | | | SEP00160 | 830817 | |
| | 200 | | | LP1423: PROCEDURE; | | | | SEP00170 | | |
| | 300 | | | /* FILE DECLARATIONS */ | | | | SEP00180 | | |
| | 400 | 2 | 1 1 | DECLARE | 1 | 2 | 3 | 4 | SEP00190 | |
| | 500 | | 1 1 | DISPFILE | FILE RECORD | SEQUENTIAL | UPDATE | ENV(INTERACTIVE); | SEP00200 830607 | |
| | 600 | | | | | | | SEP00210 | | |
| | 700 | | | /* RECORD DECLARATIONS */ | | | | SEP00220 | | |
| | 800 | 3 | 1 1 | DECLARE | | | | | SEP00230 | |
| | 900 | | 1 1 | 1 | DISPLAY_RECORD, | | | | | SEP00240 |
| | 1000 | | | %INCLUDE DISPFILE(FORMAT1,INPUT.,COMMA); | | | | SEP00250 | 831012 | |
| FORMAT1 | + 100 | | | /* ----- */ | | | | */ | | |
| FORMAT1 | + 200 | | | /* DEVICE FILE: DISPFILE.LP1423 | | | | */ | | |
| FORMAT1 | + 300 | | | /* FILE CREATION DATE: 87/11/30 | | | | */ | | |
| FORMAT1 | + 400 | | | /* RECORD FORMAT: FORMAT1 | | | | */ | | |
| FORMAT1 | + 500 | | | /* RECORD FORMAT SEQUENCE ID: 1122F3EAAD919 | | | | */ | | |
| FORMAT1 | + 600 | | | /* ----- */ | | | | */ | | |
| FORMAT1 | + 700 | | | /* INDICATORS FOR FORMAT FORMAT1 | | | | */ | | |
| FORMAT1 | + 800 | | | /* INDICATOR 01 | | | | */ | | |
| FORMAT1 | + 900 | | | /* INDICATOR 51 DAILY REPORT | | | | */ | | |
| FORMAT1 | + 1000 | | | /* INDICATOR 52 MONTHLY REPORT | | | | */ | | |
| FORMAT1 | + 1100 | | | /* INDICATOR 99 END OF PROGRAM | | | | */ | | |
| FORMAT1 | + 1200 | | | /* ----- */ | | | | */ | | |
| FORMAT1 | + 1300 | 3.1 | 1 1 | 15 | DEPTNO | CHAR(5), | | | | |
| | 1100 | 3.2 | 1 1 | 1 | INDICATOR_RECORD, | | | SEP00260 | | |
| | 1200 | | | 5 | %INCLUDE DISPFILE(FORMAT1,INDICATORS); | | | | SEP00270 830607 | |
| FORMAT1 | + 100 | | | /* ----- */ | | | | */ | | |
| FORMAT1 | + 200 | | | /* DEVICE FILE: DISPFILE.LP1423 | | | | */ | | |
| FORMAT1 | + 300 | | | /* FILE CREATION DATE: 87/11/30 | | | | */ | | |
| FORMAT1 | + 400 | | | /* RECORD FORMAT: FORMAT1 | | | | */ | | |
| FORMAT1 | + 500 | | | /* RECORD FORMAT SEQUENCE ID: 1122F3EAAD919 | | | | */ | | |
| FORMAT1 | + 600 | | | /* ----- */ | | | | */ | | |
| FORMAT1 | + 700 | 3.3 | 1 1 | 15 | IN01 | PIC '9', | | | | |
| FORMAT1 | + 800 | 3.4 | 1 1 | 15 | IN02_IN50 | CHAR(49), | /* UNDEFINED INDICATOR(S) | */ | | |
| FORMAT1 | + 900 | 3.5 | 1 1 | 15 | IN51 | PIC '9', | /* DAILY REPORT | */ | | |
| FORMAT1 | + 1000 | 3.6 | 1 1 | 15 | IN52 | PIC '9', | /* MONTHLY REPORT | */ | | |
| FORMAT1 | + 1100 | 3.7 | 1 1 | 15 | IN53_IN98 | CHAR(46), | /* UNDEFINED INDICATOR(S) | */ | | |
| FORMAT1 | + 1200 | 3.8 | 1 1 | 15 | IN99 | PIC '9'; | /* END OF PROGRAM | */ | | |
| | 1300 | | | | | | | 830607 | | |
| | 1400 | | | /* SAMPLE SUBROUTINE DECLARATIONS */ | | | | 830607 | | |
| | 1500 | 4 | 1 1 | DECLARE | | | | | 830607 | |
| | 1600 | | 1 1 | DAILY | EXTERNAL ENTRY (CHAR(5)); | | | | 830607 | |
| | 1700 | 4.1 | 1 1 | MONTHLY | EXTERNAL ENTRY (CHAR(5)); | | | | 830607 | |
| | 1800 | | | | | | | SEP00280 | | |
| | 1900 | | | /* INDICATOR FLAGS */ | | | | SEP00290 | | |
| | 2000 | 5 | 1 1 | DECLARE | | | | | SEP00300 | |
| | 2100 | | 1 1 | 1 | INDICATOR_FLAGS | STATIC, | | | SEP00310 | |
| | 2200 | 5.1 | 1 1 | 2 | OFF | PICTURE '9' INIT(0), | | | SEP00320 831012 | |
| | 2300 | 5.2 | 1 1 | 2 | ON | PICTURE '9' INIT(1); | | | SEP00330 831012 | |
| | 2400 | | | | | | | SEP00340 | | |
| | 2500 | | | /* BUILT-IN FUNCTIONS */ | | | | SEP00350 | | |
| | 2600 | 6 | 1 1 | DECLARE | | | | | SEP00360 | |
| | 2700 | | 1 1 | SUBSTR | BUILTIN, | | | | 831013 | |
| | 2800 | 6.1 | 1 1 | DATE | BUILTIN; | | | | SEP00370 | |

Figure 8-13 (Part 2 of 3). Program Using Indicators in a Separate Area


```

5728PL1 R01 M00 880715          PL/I Source Listing          LP1423/LP1423          11/30/88 15:30:01          Page   3
                                LP1423: PROCEDURE;
Include  SEQNBR STMT.SUBS BLK BN DO  *<...1...2...3...4...5...6...7...>...8 Date
2900
3000                                /* PROGRAM VARIABLES */
3100      7      1  1      DECLARE
3200                                TODAYS_DATE          CHAR(6),
3300      7.1    1  1      1  CURRENT_DATE,
3400      7.2    1  1      2  CURR_YEAR          CHAR(2),
3500      7.3    1  1      2  CURR_MONTH        CHAR(2),
3600      7.4    1  1      2  CURR_DAY          CHAR(2);
3700
3800                                /* MAIN PROGRAM */
3900      8      1  1      TODAYS_DATE = DATE;
4000      9      1  1      CURR_YEAR  = SUBSTR(TODAYS_DATE,1,2);
4100     10     1  1      CURR_MONTH  = SUBSTR(TODAYS_DATE,3,2);
4200     11     1  1      CURR_DAY   = SUBSTR(TODAYS_DATE,5,2);
4300     12     1  1      IN99 = OFF;
4400
4500     13     1  1      6  OPEN
4600     13     1  1      FILE (DISPFILE); /* UPDATE */
4700
4800     14     1  1      7  DO WHILE(IN99 = OFF);
4900
5000                                /* DISPLAY THE SCREEN */
5100     15     1  1  1      IN01 = OFF;
5200     16     1  1  1      IF CURR_DAY = '01' THEN
5300     16     1  1  1      IN01 = ON;
5400     17     1  1  1      8  WRITE FILE (DISPFILE) FROM (DISPLAY_RECORD)
5500     17     1  1  1      OPTIONS (RECORD('FORMAT1') INDICATORS(INDICATOR_RECORD));
5600
5700                                /* READ AND PROCESS SCREEN */
5800     18     1  1  1      INDICATOR_RECORD = OFF;
5900     19     1  1  1      9  READ FILE (DISPFILE) INTO (DISPLAY_RECORD)
6000     19     1  1  1      OPTIONS (RECORD('FORMAT1') INDICATORS(INDICATOR_RECORD));
6100     20     1  1  1      10 IF IN51 = ON THEN
6200     20     1  1  1      CALL DAILY(DEPTNO); /* EXTERNAL PROCEDURE */
6300     21     1  1  1      10 ELSE
6400     21     1  1  1      IF IN52 = ON THEN
6500     21     1  1  1      CALL MONTHLY(DEPTNO); /* EXTERNAL PROCEDURE */
6600     22     1  1  1      END; /* DO WHILE */
6700
6800     23     1  1      CLOSE
6900     23     1  1      FILE (DISPFILE);
7000
7100     24     1  1      END LP1423;

```

Figure 8-13 (Part 3 of 3). Program Using Indicators in a Separate Area

USING DISPLAY FILES

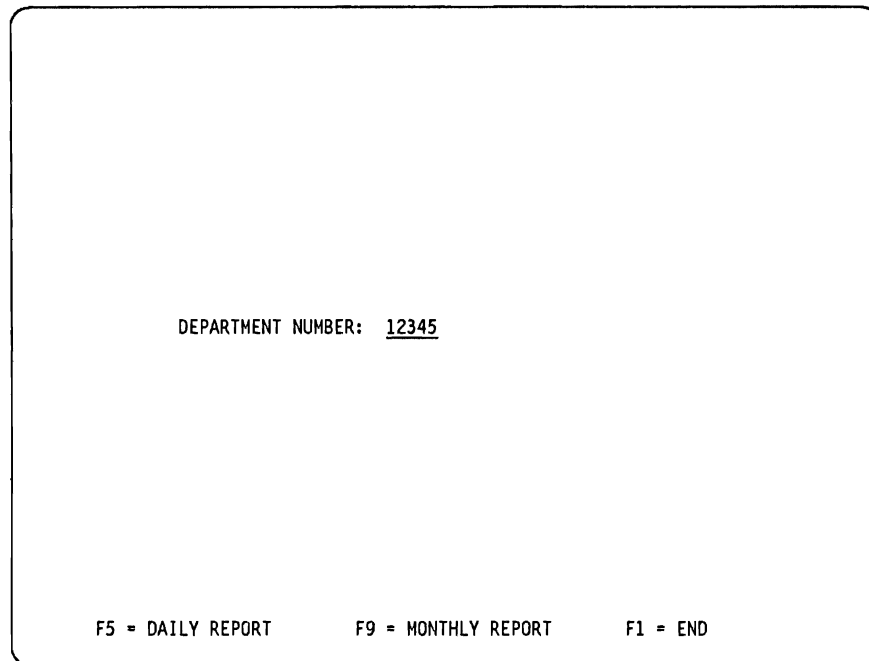


Figure 8-14. Display for Program Using Indicators in a Separate Area

- 1** RECORD data transmission is used with DISPFILE.
- 2** The SEQUENTIAL access method is used with DISPFILE.
- 3** The data transmission mode is specified as UPDATE, because the file is used for both input and output.
- 4** The ENVIRONMENT attribute is specified with the INTERACTIVE option: both READ and WRITE statements are processed.
- 5** The %INCLUDE statement generates declarations for the INDICATORS specified in record format FORMAT1 of file DISPFILE.
- 6** DISPFILE is opened. The data transmission mode UPDATE is specified in the record declaration. It could also be specified here as an option of the OPEN statement.
- 7** The program continues running as long as IN99 has a value of OFF, that is, until the user ends program running.
- 8** The prompt screen is displayed by writing DISPLAY_RECORD to DISPFILE. Record format FORMAT1 is used and the indicators passed to the display device are those declared in INDICATOR_RECORD.
- 9** The information provided by the user is read into DISPLAY_RECORD from file DISPFILE using record format FORMAT1, and the values of the indicators in INDICATOR_RECORD are returned by the device.
- 10** Depending on the values of IN51 and IN52, an appropriate external procedure is called.

Program Using Program-Defined Indicators in a Separate Area: The following program defines the same DDS as the above example using a separate indicator area. The indicators in this example are program-described.

```

5728PL1 R01 M00 880715          PL/I Source Listing          PLITST/LP1424          11/30/88 14:48:47          Page 2
                                LP1424: PROCEDURE;          PGM00160
Include      SEQNBR STMT.SUBS BLK BN DO  *<.....1.....2.....3.....4.....5.....6.....7.....8 Date
                                LP1424: PROCEDURE;          PGM00160 830817
                                100      1          PGM00170
                                200          PGM00180
                                300          PGM00190
                                400      2      1 1      /* FILE DECLARATIONS */          PGM00200 830607
                                500      1 1      DECLARE          1 2 5 4          PGM00210
                                DISPFILE FILE RECORD SEQUENTIAL UPDATE ENV(INTERACTIVE);          PGM00220
                                600          PGM00230
                                700          /* RECORD DECLARATIONS */          PGM00240
                                800      3      1 1      DECLARE          PGM00250 831012
                                900      1 1      1 DISPLAY_RECORD,
                                1000     %INCLUDE DISPFILE(FORMAT1,INPUT,,COMMA);
FORMAT1     + 100          /* ----- */          /*
FORMAT1     + 200          /* * DEVICE FILE: DISPFILE.DBLIB          /*
FORMAT1     + 300          /* * FILE CREATION DATE: 87/11/05          /*
FORMAT1     + 400          /* * RECORD FORMAT: FORMAT1          /*
FORMAT1     + 500          /* * RECORD FORMAT SEQUENCE ID: 1122F3EAAD919          /*
FORMAT1     + 600          /* ----- */          /*
FORMAT1     + 700          /* * INDICATORS FOR FORMAT FORMAT1          /*
FORMAT1     + 800          /* * INDICATOR 01          /*
FORMAT1     + 900          /* * INDICATOR 51          DAILY REPORT          /*
FORMAT1     + 1000         /* * INDICATOR 52          MONTHLY REPORT          /*
FORMAT1     + 1100        /* * INDICATOR 99          END OF PROGRAM          /*
FORMAT1     + 1200        /* ----- */          /*
FORMAT1     + 1300        3.1      1 1          15 DEPTNO          CHAR(5),
                                1100        3.2      1 1      5 1 INDICATOR_RECORD,          PGM00260
                                1200        3.3      1 1          15 IND_NEW_MONTH          PICTURE '9', /* IN01 NEW MONTH */ PGM00270
                                1300        3.4      1 1          15 IN02_IN50          CHAR(49), /* UNDEF INDICATORS */ PGM00280
                                1400        3.5      1 1          15 IND_DAILY          PICTURE '9', /* IN51 DAILY REPRT */ PGM00290
                                1500        3.6      1 1          15 IND_MONTHLY          PICTURE '9', /* IN52 MONTHLY RPRT*/ PGM00300
                                1600        3.7      1 1          15 IN53_IN98          CHAR(46), /* UNDEF INDICATORS */ PGM00310
                                1700        3.8      1 1          15 IND_EOJ          PICTURE '9'; /* IN99 END OF JOB */ PGM00320
                                1800
                                1900          /* * SAMPLE SUBROUTINE DECLARATIONS */          830607
                                2000        4      1 1      DECLARE          830607
                                2100          DAILY          EXTERNAL ENTRY (CHAR(5)),          830607
                                2200        4.1      1 1          MONTHLY          EXTERNAL ENTRY (CHAR(5));          830607
                                2300
                                2400          /* * INDICATOR FLAGS */          PGM00330
                                2500        5      1 1      DECLARE          PGM00340
                                2600          1 INDICATOR_FLAGS STATIC,          PGM00350
                                2700          2 OFF          PICTURE '9' INIT(0),          PGM00360
                                2800          2 ON          PICTURE '9' INIT(1);          PGM00370 831012
                                2900          PGM00380 831012
                                3000          /* * BUILT-IN FUNCTIONS */          PGM00390
                                3100        6      1 1      DECLARE          PGM00400
                                3200          SUBSTR          BUILTIN,          PGM00410
                                3300        6.1      1 1          DATE          BUILTIN;          831013
                                3400          PGM00420 831013
                                3500          /* * PROGRAM VARIABLES */          PGM00430
                                3600        7      1 1      DECLARE          PGM00440
                                3700          TODAYS_DATE          CHAR(6),          PGM00450
                                3800          1 CURRENT_DATE,          831013
                                3900          2 CURR_YEAR          CHAR(2),          PGM00460
                                4000          2 CURR_MONTH          CHAR(2),          PGM00470
                                PGM00480

```

Figure 8-15 (Part 1 of 2). Program Using Program-Defined Indicators in a Separate Area

USING DISPLAY FILES

| 5728PL1 R01 M00 880715 | | | | | PL/I Source Listing | PLITST/LP1424 | 11/30/88 14:48:47 | Page 3 |
|------------------------|--------|-----------|--------|----|--|---------------|-------------------|-----------------|
| Include | SEQNBR | STMT.SUBS | BLK BN | DO | *<...1...+...2...+...3...+...4...+...5...+...6...+...7...>...8 | Date | | |
| | 4100 | 7.4 | 1 1 | | 2 CURR_DAY | CHAR(2); | | PGM00160 |
| | 4200 | | | | | | | PGM00490 |
| | 4300 | | | | | | | PGM00500 |
| | 4400 | 8 | 1 1 | 6 | /* MAIN PROGRAM */ | | | PGM00510 |
| | 4500 | | 1 1 | | OPEN | | | PGM00520 |
| | 4600 | | | | FILE (DISPFILE); /* UPDATE */ | | | PGM00530 830919 |
| | 4700 | 9 | 1 1 | | | | | PGM00540 |
| | 4800 | 10 | 1 1 | | TODAYS_DATE = DATE; | | | PGM00550 831013 |
| | 4900 | 11 | 1 1 | | CURR_YEAR = SUBSTR(TODAYS_DATE,1,2); | | | 831013 |
| | 5000 | 12 | 1 1 | | CURR_MONTH = SUBSTR(TODAYS_DATE,3,2); | | | 831013 |
| | 5100 | 13 | 1 1 | | CURR_DAY = SUBSTR(TODAYS_DATE,5,2); | | | 831013 |
| | 5200 | 14 | 1 1 | 7 | IND_EOJ = OFF; | | | PGM00560 |
| | 5300 | | | | DO WHILE(IND_EOJ = OFF); | | | PGM00570 830608 |
| | 5400 | | | | | | | PGM00580 |
| | 5500 | 15 | 1 1 1 | | /* DISPLAY THE SCREEN */ | | | PGM00590 |
| | 5600 | 16 | 1 1 1 | | INDICATOR_RECORD = OFF; | | | PGM00600 |
| | 5700 | | 1 1 1 | | IF CURR_DAY = '01' THEN | | | PGM00610 |
| | 5800 | 17 | 1 1 1 | 8 | IND_NEW_MONTH = ON; | | | PGM00620 |
| | 5900 | | 1 1 1 | | WRITE FILE (DISPFILE) FROM (DISPLAY_RECORD) | | | PGM00630 830607 |
| | 6000 | | | | OPTIONS (RECORD('FORMAT1') INDICATORS(INDICATOR_RECORD)); | | | PGM00640 831013 |
| | 6100 | | | | | | | PGM00650 |
| | 6200 | 18 | 1 1 1 | 9 | /* READ AND PROCESS SCREEN */ | | | PGM00660 |
| | 6300 | | 1 1 1 | | READ FILE (DISPFILE) INTO (DISPLAY_RECORD) | | | PGM00670 830607 |
| | 6400 | 19 | 1 1 1 | 10 | OPTIONS (RECORD('FORMAT1') INDICATORS(INDICATOR_RECORD)); | | | PGM00680 831013 |
| | 6500 | | 1 1 1 | | IF IND_DAILY = ON THEN | | | PGM00690 |
| | 6600 | 20 | 1 1 1 | 10 | CALL DAILY(DEPTNO); | | | PGM00700 |
| | 6700 | | 1 1 1 | | ELSE | | | PGM00710 |
| | 6800 | | 1 1 1 | | IF IND_MONTHLY = ON THEN | | | PGM00720 |
| | 6900 | 21 | 1 1 1 | | CALL MONTHLY(DEPTNO); | | | PGM00730 |
| | 7000 | | | | END; /* DO WHILE */ | | | PGM00740 |
| | 7100 | 22 | 1 1 | | | | | PGM00750 |
| | 7200 | | 1 1 | | CLOSE | | | PGM00760 |
| | 7300 | | | | FILE (DISPFILE); | | | PGM00770 830607 |
| | 7400 | 23 | 1 1 | | END LP1424; | | | PGM00780 |
| | | | | | | | | PGM00790 830817 |

Figure 8-15 (Part 2 of 2). Program Using Program-Defined Indicators in a Separate Area

- 1** RECORD data transmission is used with DISPFILE.
- 2** The SEQUENTIAL access method is used with DISPFILE.
- 3** The data transmission mode is specified as UPDATE, because the file is used for both input and output.
- 4** The ENVIRONMENT attribute is specified with the INTERACTIVE option: both READ and WRITE statements are processed.
- 5** The %INCLUDE statement is not used to generate declarations for the indicators. Instead, the indicators are program-defined. This has the advantage that you can give the indicators meaningful variable names. It has the disadvantage that you must ensure that you have coded the indicators correctly. The %INCLUDE statement generates declarations for the INPUT fields specified in record format FORMAT1 of file DISPFILE.
- 6** DISPFILE is opened. The data transmission mode UPDATE is specified in the record declaration. It could also be specified here as an option of the OPEN statement.
- 7** The program continues running as long as IND_EOJ has a value of OFF, that is, until the user ends program running.

- 8** The prompt screen is displayed by writing DISPLAY_RECORD to DISPPFILE. Record format FORMAT1 is used and the indicators passed to the display device are those declared in INDICATOR_RECORD.
- 9** The information provided by the user is read into DISPLAY_RECORD from file DISPPFILE using record format FORMAT1, and the values of the indicators in INDICATOR_RECORD are returned by the device.
- 10** Depending on the values of IND_DAILY and IND_MONTHLY an appropriate external procedure is called.

Using Indicators in the Record Area: You can also use indicators that are included in your record area. To do this, you must specify NOINDARA as an option in the ENVIRONMENT attribute of the file declaration. You can even use %INCLUDE to include the indicators, but they will not be in a separate area. This method of using indicators is not recommended. It is available in AS/400 PL/I to ensure compatibility if you are translating a program written in another AS/400 language.

When using indicators in the record buffer you must take into consideration the fact that the indicators may not be in ascending sequential order. The indicators are located in the buffer in the order in which they are declared in the DDS source for the file. For example, if indicator 97 is the first indicator declared in the DDS for the file, it will also be the first in order in the buffer.

If you are using externally described record definitions, the %INCLUDE directive will list the indicators according to their order in the input/output buffers. If the record definitions are program-described, you must ensure that the order in which you list the indicators is the same as the order they have in the buffer. If you fail to do this, there may be a mismatch between the indicators set on in the program and the indicators set on in the file description.

Using Device Files

Although the different types of device files (Communications, BSC, Printer, Inline, Tape, and Diskette) have some differences between them, there are also some similarities. Most importantly, they all support the combination of CONSECUTIVE organization and SEQUENTIAL access. This is to increase device independence and allow for file redirection. With Communications and BSC files, INTERACTIVE organization is also allowed. The input/output statements allowed with each file type are shown in Appendix C, "Valid Combinations of Options for Input/Output Statements."

Other ENVIRONMENT options which can be specified for device files are BUFSIZE, which can be specified for any type of file; BLOCK, which is valid for Tape and Diskette files only; and NOINDARA, which can be specified for Communications, BSC, and Printer files. All of these are discussed in Chapter 7, "File Declaration and Input/Output."

For TAPE files with variable length records or undefined records, the length of the variable on the WRITE statement determines the length of the tape record.

USING DEVICE FILES

Externally Described Device Files

Communications, BSC, display and printer files are the only device files that can be externally described. For these files, you can specify the DESCRIBED option, as discussed in Chapter 7, "File Declaration and Input/Output."

Program-Described Device Files

All types of device files can be program-described. If a printer file is program-described, then you have the option of using the CTLASA option, described in Chapter 7, "File Declaration and Input/Output."

Example of Using a Printer File

The following program and supporting DDS uses a printer file to create a report.

IBM International Business Machines **AS/400 DATA DESCRIPTION SPECIFICATIONS** 6X21-9891-0 UM/056-
Printed in U.S.A.
*Number of sheets per pad may vary slightly.

| | | | | |
|------------|--------------------|-------------|-------------|---------|
| File | Keying Instruction | Graphic Key | Description | Page of |
| Programmer | Date | | | |

| Sequence Number | Form Type 1 And/Or Comment (A/D/n) | Indicator | Indicator | Indicator | Indicator | Type of Name of Sys. (B/R/T/L/S/D) | Name | Reference (R) | Length | Location | | Function |
|-----------------|---------------------------------------|-----------|-----------|-----------|-----------|------------------------------------|--|---------------|--------|----------|-----|-----------------------------|
| | | | | | | | | | | Line | Pos | |
| A* | | | | | | | PHYSICAL FILE DDS FOR PERSONNEL FILE IN PRINTER FILE EXAMPLE | | | | | |
| A | | | | | | | R PERSREC | | | | | |
| A | | | | | | | EMPLNO | | 6S | | | TEXT('EMPLOYEE NUMBER') |
| A | | | | | | | NAME | | 28 | | | TEXT('EMPLOYEE NAME') |
| A | | | | | | | ADDRESS1 | | 35 | | | TEXT('HOME ADDRESS') |
| A | | | | | | | ADDRESS2 | | 35 | | | TEXT('SPOUSE WORK ADDRESS') |
| A | | | | | | | BIRTHDATE | | 6 | | | TEXT('DATE OF BIRTH') |
| A | | | | | | | MARSTAT | | 1 | | | TEXT('MARITAL STATUS') |
| A | | | | | | | SPOUSENAME | | 28 | | | TEXT('NAME OF SPOUSE') |
| A | | | | | | | NUMCHILD | | 25 | | | TEXT('NUMBER OF CHILDREN') |
| A | | | | | | | K EMPLNO | | | | | |
| A | | | | | | | | | | | | |
| A | | | | | | | | | | | | |
| A | | | | | | | | | | | | |
| A | | | | | | | | | | | | |
| A | | | | | | | | | | | | |
| A | | | | | | | | | | | | |

Figure 8-16 (Part 1 of 6). Program Using Printer File and Supporting DDS

AS/400 DATA DESCRIPTION SPECIFICATIONS

| File | | Keying Instruction | Graphic | Description | Page | of |
|------------|--|--------------------|---------|-------------|------|----|
| Programmer | | Date | Key | | | |

| Sequence Number | Conditioning | | | | Name | Length | Reference (R) | Location | | Functions |
|-----------------|--|-----------|-----------|-----------|----------------------|--------|---------------|----------|-----|-----------------------|
| | Form Type 1-4 5-8 9-12 13-16 17-18 19-20 21-22 23-24 25-26 27-28 29-30 31-32 33-34 35 36 37 38 39-40 41-42 43-44 45-46 47-48 49-50 51-52 53-54 55-56 57-58 59-60 61-62 63-64 65-66 67-68 69-70 71-72 73-74 75-76 77-78 79-80 | Indicator | Indicator | Indicator | | | | Line | Pos | |
| A* | PRINTER FILE DDS FOR PERSONNEL FILE | | | | | | | | | |
| A | | | | R | INDARA REF(PERSFILE) | | | | | |
| A | | | | R | HEADING | | | | | SKIPB(1) SPACEA(3) |
| A | | | | | | | | | | 15'PERSONNEL LISTING' |
| A | | | | | | | | | | UNDERLINE |
| A | | | | | | | | | | 33'- ORDERED BY' |
| A | | | | | ORDERTYPE | 33 | | | | 46 |
| A | | | | | | | | | | 80DATE EDTCDE(Y) |
| A | | | | | | | | | | 93TIME |
| A | | | | | | | | | | 115'PAGE:' |
| A | | | | | | | | | | +1PAGNBR EDTCDE(J) |
| A* | LINE 1 | | | | | | | | | |
| A | | | | R | DETAIL | | | | | SPACEA(3) |
| A | | | | | | | | | | 1'NAME:' |
| A | | | | | NAME | R | | | | 11UNDERLINE |
| A | | | | | | | | | | 55'EMPLOYEE NUMBER:' |
| A | | | | | EMPLNO | R | | | | 73 |
| A | | | | | | | | | | 87'DATE OF BIRTH:' |
| A | | | | | BIRTHDATE | R | | | | 103SPACEA(1) |

Figure 8-16 (Part 2 of 6). Program Using Printer File and Supporting DDS

AS/400 DATA DESCRIPTION SPECIFICATIONS

| File | | Keying Instruction | Graphic | Description | Page | of |
|------------|--|--------------------|---------|-------------|------|----|
| Programmer | | Date | Key | | | |

| Sequence Number | Conditioning | | | | Name | Length | Reference (R) | Location | | Functions |
|-----------------|--|-----------|-----------|-----------|-------------|--------|---------------|----------|-----|---------------------|
| | Form Type 1-4 5-8 9-12 13-16 17-18 19-20 21-22 23-24 25-26 27-28 29-30 31-32 33-34 35 36 37 38 39-40 41-42 43-44 45-46 47-48 49-50 51-52 53-54 55-56 57-58 59-60 61-62 63-64 65-66 67-68 69-70 71-72 73-74 75-76 77-78 79-80 | Indicator | Indicator | Indicator | | | | Line | Pos | |
| A* | LINE 2 | | | | | | | | | |
| A | | | | | | | | | | 1'ADDRESS:' |
| A | | | | | ADDRESS1 | R | | | | 11 |
| A | | | | | | | | | | 55'MARITAL STATUS:' |
| A | | | | | MARSTAT | R | | | | 73 |
| A | 01 | | | | | | | | | 87'SPOUSE''S NAME' |
| A | 01 | | | | SPOUSENAMER | | | | | 103 |
| A* | LINE 3 | | | | | | | | | |
| A | | | | | ADDRESS2 | R | | | | 11SPACEB(1) |
| A | | | | | | | | | | 55'CHILDREN:' |
| A | | | | | NUMCHILD | R | | | | 72EDTCDE(3) |

Figure 8-16 (Part 3 of 6). Program Using Printer File and Supporting DDS

USING DEVICE FILES

```

5728PL1 R01 M00 880715          PL/I Source Listing          QTEMP/LP1421          11/30/88 15:37:16          Page 2
LP1421: PROCEDURE;
*-<.....1.....2.....3.....4.....5.....6.....7.....>.....8 Date
LP1421: PROCEDURE;
/* FILE DECLARATIONS */
DECLARE          1          2          3          4          5
PERSFILE FILE RECORD SEQUENTIAL INPUT ENV(INDEXED DESCRIBED),
PERSREPT FILE RECORD SEQUENTIAL OUTPUT ENV(CONSECUTIVE DESCRIBED);
/* RECORD DECLARATIONS */
DECLARE
1 PERSFILE_RECORD,
  %INCLUDE PERSFILE(PERSREC,RECORD,,COMMA);
PERSREC + 100 /* ----- */
PERSREC + 200 /* PHYSICAL FILE: PERSFILE.QTEMP */
PERSREC + 300 /* FILE CREATION DATE: 87/11/30 */
PERSREC + 400 /* RECORD FORMAT: PERSREC */
PERSREC + 500 /* RECORD FORMAT SEQUENCE ID: 58986345F0578 */
PERSREC + 600 /* ----- */
PERSREC + 700 3.1 1 1 15 EMLNO PIC '99999R', /* EMPLOYEE NUMBER */
PERSREC + 800 /* DDS - KEY FIELD */
PERSREC + 900 3.2 1 1 15 NAME CHAR(28), /* EMPLOYEE NAME */
PERSREC + 1000 3.3 1 1 15 ADDRESS1 CHAR(35), /* HOME ADDRESS */
PERSREC + 1100 3.4 1 1 15 ADDRESS2 CHAR(35), /* SPOUSE WORK ADDRESS */
PERSREC + 1200 3.5 1 1 15 BIRTHDATE CHAR(6), /* DATE OF BIRTH */
PERSREC + 1300 3.6 1 1 15 MARSTAT CHAR(1), /* MARITAL STATUS */
PERSREC + 1400 3.7 1 1 15 SPOUSENAME CHAR(28), /* NAME OF SPOUSE */
PERSREC + 1500 3.8 1 1 15 NUMCHILD PIC '9R', /* NUMBER OF CHILDREN */
1200 3.9 1 1 1 HEADING_OUTPUT,
1300 %INCLUDE PERSREPT(HEADING,OUTPUT,,COMMA);
HEADING + 100 /* ----- */
HEADING + 200 /* DEVICE FILE: PERSREPT.QTEMP */
HEADING + 300 /* FILE CREATION DATE: 87/11/30 */
HEADING + 400 /* RECORD FORMAT: HEADING */
HEADING + 500 /* RECORD FORMAT SEQUENCE ID: 1450B069F8720 */
HEADING + 600 /* ----- */
HEADING + 700 3.10 1 1 15 ORDERTYPE CHAR(33),
1400 3.11 1 1 1 DETAIL_OUTPUT,
1500 %INCLUDE PERSREPT(DETAIL,OUTPUT,,COMMA);
DETAIL + 100 /* ----- */
DETAIL + 200 /* DEVICE FILE: PERSREPT.QTEMP */
DETAIL + 300 /* FILE CREATION DATE: 87/11/30 */
DETAIL + 400 /* RECORD FORMAT: DETAIL */
DETAIL + 500 /* RECORD FORMAT SEQUENCE ID: 1EC226084A4DB */
DETAIL + 600 /* ----- */
DETAIL + 700 /* INDICATORS FOR FORMAT DETAIL */
DETAIL + 800 /* INDICATOR 01 */
DETAIL + 900 /* ----- */
DETAIL + 1000 3.12 1 1 15 NAME CHAR(28), /* EMPLOYEE NAME */
DETAIL + 1100 3.13 1 1 15 EMLNO PIC '99999R', /* EMPLOYEE NUMBER */
DETAIL + 1200 3.14 1 1 15 BIRTHDATE CHAR(6), /* DATE OF BIRTH */
DETAIL + 1300 3.15 1 1 15 ADDRESS1 CHAR(35), /* HOME ADDRESS */
DETAIL + 1400 3.16 1 1 15 MARSTAT CHAR(1), /* MARITAL STATUS */
DETAIL + 1500 3.17 1 1 15 SPOUSENAME CHAR(28), /* NAME OF SPOUSE */
DETAIL + 1600 3.18 1 1 15 ADDRESS2 CHAR(35), /* SPOUSE WORK ADDRESS */

```

Figure 8-16 (Part 4 of 6). Program Using Printer File and Supporting DDS

| 5728PL1 R01 M00 880715 | | PL/I Source Listing | | QTEMP/LP1421 | 11/30/88 15:37:16 | Page 3 |
|------------------------|--------|---------------------|-----------|---|-------------------|-----------------|
| Include | SEQNBR | STMT.SUBS | BLK BN DO | LP1421: PROCEDURE; | | PRI00160 |
| DETAIL | + 1700 | 3.19 | 1 1 | *<.....1.....2.....3.....4.....5.....6.....7.>.....8 Date | | |
| | 1600 | 3.20 | 1 1 | 15 NUMCHILD PIC '9R', /* NUMBER OF CHILDREN */ | | |
| | 1700 | | | 1 PERSREPT_INDICATORS, PRI00310 | | |
| DETAIL | + 100 | | | %INCLUDE PERSREPT(DETAIL,INDICATORS); PRI00320 | | |
| DETAIL | + 200 | | | /* ----- */ | | |
| DETAIL | + 300 | | | /* DEVICE FILE: PERSREPT.QTEMP */ | | |
| DETAIL | + 400 | | | /* FILE CREATION DATE: 87/11/30 */ | | |
| DETAIL | + 500 | | | /* RECORD FORMAT: DETAIL */ | | |
| DETAIL | + 600 | | | /* RECORD FORMAT SEQUENCE ID: 1EC226084A4DB */ | | |
| DETAIL | + 700 | 3.21 | 1 1 | 15 IN01 PIC '9', | | |
| DETAIL | + 800 | 3.22 | 1 1 | 15 IN02_IN99 CHAR(98); /* UNDEFINED INDICATOR(S) */ | | |
| | 1800 | | | | | PRI00330 |
| | 1900 | | | /* INDICATOR FLAGS */ | | PRI00340 |
| | 2000 | 4 | 1 1 | DECLARE | | PRI00350 |
| | 2100 | | 1 1 | 1 INDICATOR_FLAGS STATIC, | | PRI00360 |
| | 2200 | 4.1 | 1 1 | 2 OFF PICTURE '9' INIT(0), | | PRI00370 831006 |
| | 2300 | 4.2 | 1 1 | 2 ON PICTURE '9' INIT(1); | | PRI00380 831006 |
| | 2400 | | | | | PRI00390 |
| | 2500 | | | /* PROGRAM FLAGS */ | | PRI00400 |
| | 2600 | 5 | 1 1 | DECLARE | | PRI00410 |
| | 2700 | | 1 1 | 1 BIT_FLAGS STATIC, | | PRI00420 |
| | 2800 | 5.1 | 1 1 | 2 MORE_RECORDS BIT(1) ALIGNED, | | PRI00430 |
| | 2900 | 5.2 | 1 1 | 2 NO BIT(1) ALIGNED INIT('0'B), | | PRI00440 |
| | 3000 | 5.3 | 1 1 | 2 YES BIT(1) ALIGNED INIT('1'B); | | PRI00450 |
| | 3100 | | | | | PRI00460 |
| | 3200 | | | /* STATIC VARIABLES */ | | PRI00470 |
| | 3300 | 6 | 1 1 | DECLARE | | PRI00480 |
| | 3400 | | 1 1 | 1 STATIC_VARIABLES STATIC, | | PRI00490 |
| | 3500 | 6.1 | 1 1 | 2 HEAD_FMT CHAR(10) INIT('HEADING'), | | PRI00500 |
| | 3600 | 6.2 | 1 1 | 2 HEAD_ORDER CHAR(15) INIT('EMPLOYEE NUMBER'), | | PRI00520 |
| | 3700 | 6.3 | 1 1 | 2 DETAIL_FMT CHAR(10) INIT('DETAIL'), | | PRI00530 |
| | 3800 | 6.4 | 1 1 | 2 DETAIL_LINES BINARY FIXED(5) INIT(5), | | PRI00540 831006 |
| | 3900 | 6.5 | 1 1 | 2 MARRIED CHAR(1) INIT('M'), | | PRI00550 |
| | 4000 | 6.6 | 1 1 | 2 PAGE_SIZE BINARY FIXED(7) INIT(50); | | PRI00560 831006 |
| | 4100 | | | | | PRI00570 |
| | 4200 | | | /* PROGRAM VARIABLES */ | | PRI00580 |
| | 4300 | 7 | 1 1 | DECLARE | | PRI00590 |
| | 4400 | | 1 1 | LINE_COUNT BINARY FIXED(5); | | PRI00600 831007 |
| | 4500 | | | | | PRI00610 |
| | 4600 | 8 | 1 1 | ON ENDFILE(PERSFILE) | | PRI00620 |
| | 4700 | | 1 1 | MORE_RECORDS = NO; | | PRI00630 |
| | 4800 | | | | | PRI00640 |
| | 4900 | | | /* MAIN PROGRAM */ | | PRI00650 |
| | 5000 | 9 | 1 1 | MORE_RECORDS = YES; | | PRI00660 |
| | 5100 | | | | | PRI00680 |
| | 5200 | 10 | 1 1 | 10 OPEN | | PRI00690 |
| | 5300 | | 1 1 | FILE (PERSFILE); /* INPUT */ | | PRI00700 830919 |
| | 5400 | 11 | 1 1 | 10 OPEN | | PRI00710 |
| | 5500 | | 1 1 | FILE (PERSREPT); /* OUTPUT */ | | PRI00720 830919 |
| | 5600 | | | | | PRI00730 |
| | 5700 | 12 | 1 1 | READ FILE (PERSFILE) INTO (PERSFILE_RECORD); | | PRI00740 831007 |
| | 5800 | | | | | PRI00750 |
| | 5900 | 13 | 1 1 | DO WHILE(MORE_RECORDS); | | PRI00760 |

Figure 8-16 (Part 5 of 6). Program Using Printer File and Supporting DDS

USING DEVICE FILES

| 5728PL1 R01 M00 880715 | | PL/I Source Listing | | QTEMP/LP1421 | | 11/30/88 15:37:16 | | Page 4 | |
|------------------------|--------|---------------------|--------|--------------|--|-------------------|----|----------|-----------------|
| | | LP1421: PROCEDURE; | | | | | | PRI00160 | |
| Include | SEQNBR | STMT.SUBS | BLK BN | DO | *<...+...1...+...2...+...3...+...4...+...5...+...6...+...7...>...+...8 | Date | | | |
| | 6000 | | | | | | | | PRI00770 |
| | 6100 | 14 | 1 1 1 | | ORDERTYPE = HEAD_ORDER; | | | | PRI00800 830919 |
| | 6200 | 15 | 1 1 1 | 11 | WRITE FILE (PERSREPT) FROM (HEADING_OUTPUT) | | | | PRI00810 830919 |
| | 6300 | | 1 1 1 | | OPTIONS(RECORD('HEADING')); | | | | PRI00820 831007 |
| | 6400 | | | | | | | | PRI00850 |
| | 6500 | 16 | 1 1 1 | | DO LINE_COUNT = 1 TO PAGE_SIZE; | | | | 831007 |
| | 6600 | 17 | 1 1 2 | | DETAIL_OUTPUT = PERSFILE_RECORD, BY NAME; | | | | PRI00860 830919 |
| | 6700 | 18 | 1 1 2 | | IF PERSFILE_RECORD.MARSTAT = MARRIED THEN | | | | PRI00870 830919 |
| | 6800 | | 1 1 2 | | IN01 = ON; | | | | PRI00880 830919 |
| | 6900 | 19 | 1 1 2 | | ELSE | | | | PRI00890 830919 |
| | 7000 | | 1 1 2 | | IN01 = OFF; | | | | PRI00900 830919 |
| | 7100 | | | | | | | | PRI00910 |
| | 7200 | 20 | 1 1 2 | 12 | WRITE FILE (PERSREPT) FROM (DETAIL_OUTPUT) | | | | PRI00920 830919 |
| | 7300 | | 1 1 2 | | OPTIONS(RECORD('DETAIL') INDICATORS(PERSREPT_INDICATORS)); | | | | PRI00930 831007 |
| | 7400 | 21 | 1 1 2 | | LINE_COUNT = LINE_COUNT + DETAIL_LINES; | | 13 | | PRI00940 830919 |
| | 7500 | 22 | 1 1 2 | | READ FILE (PERSFILE) INTO (PERSFILE_RECORD); | | | | PRI00950 831007 |
| | 7600 | 23 | 1 1 2 | | IF ~MORE_RECORDS THEN | | | | 831007 |
| | 7700 | | 1 1 2 | | LINE_COUNT = PAGE_SIZE + 1; | | | | 830919 |
| | 7800 | 24 | 1 1 2 | | END; /* DO */ | | | | 830919 |
| | 7900 | | | | | | | | 830919 |
| | 8000 | 25 | 1 1 1 | | END; /* DO WHILE */ | | | | PRI00960 |
| | 8100 | | | | | | | | PRI00970 |
| | 8200 | 26 | 1 1 | | CLOSE | | | | PRI00980 |
| | 8300 | | 1 1 | | FILE (PERSFILE); | | | | PRI00990 |
| | 8400 | 27 | 1 1 | | CLOSE | | | | PRI01000 |
| | 8500 | | 1 1 | | FILE (PERSREPT); | | | | PRI01010 |
| | 8600 | | | | | | | | PRI01020 |
| | 8700 | 28 | 1 1 | | END LP1421; | | | | PRI01030 830817 |

Figure 8-16 (Part 6 of 6). Program Using Printer File and Supporting DDS

- 1 RECORD data transmission is used with PERSFILE.
- 2 The SEQUENTIAL access method is used with PERSFILE.
- 3 PERSFILE is used for INPUT only; no data is directed to it.
- 4 INDEXED specifies that PERSFILE is processed using the keyed sequence access path.
- 5 DESCRIBED specifies that the %INCLUDE directive is used to bring external record format definitions from PERSFILE into the program. The DESCRIBED option ensures that at the time of program compilation the PL/I file attributes match the system file attributes, and that level checking is processed when the file is opened during program running.
- 6 RECORD data transmission is used with PERSREPT.
- 7 The SEQUENTIAL access method is used with PERSREPT.
- 8 PERSREPT is used for OUTPUT only. It does not provide data to the program.
- 9 The CONSECUTIVE option specifies that the file is processed using the arrival sequence access path. Because CONSECUTIVE is the default, the CONSECUTIVE option could have been omitted.
- 10 PERSFILE and PERSREPT are opened. Their data transmission modes (INPUT and OUTPUT) are omitted here, because they are included as attributes in the file declaration.

- 11** The first line printed on each page is the heading. The first WRITE statement writes record `HEADING_OUTPUT` to file `PERSREPT` using record format `HEADING`.
- 12** Record `DETAIL_OUTPUT` is written to `PERSREPT` using record format `DETAIL`.
- 13** The indicators in `PERSREPT_INDICATORS` are passed to the printer.

Using STREAM Files

Stream files are a type of program-described file. They can be connected only to files that do not have record definitions. They are file-independent because they can only be used for arrival sequence access.

USING STREAM FILES

Example of Using a Stream File

| 5728PL1 R01 M00 880715 | | PL/I Source Listing | | PLITST/LP1418 | 11/30/88 14:21:11 | Page 2 |
|------------------------|--------|---------------------|-----------|---|----------------------------|-----------------|
| Include | SEQNBR | STMT.SUBS | BLK BN DO | LP1418: PROCEDURE; | | STR00160 |
| | 100 | 1 | | *<.....1.....2.....3.....4.....5.....6.....7.>.....8 Date | | STR00160 830817 |
| | 200 | | | LP1418: PROCEDURE; | | STR00170 |
| | 300 | | | /* FILE DECLARATIONS */ | | STR00180 |
| | 400 | 2 | 1 1 | DECLARE | 1 2 | STR00190 |
| | 500 | | 1 1 | EMP_FILE FILE STREAM INPUT, | | STR00200 |
| | 600 | 2.1 | 1 1 | SYSPRINT FILE STREAM OUTPUT PRINT; | | STR00210 |
| | 700 | | | 3 4 5 | | STR00220 |
| | 800 | | | /* VARIABLE DECLARATIONS */ | | STR00230 |
| | 900 | 3 | 1 1 | DECLARE | | STR00240 |
| | 1000 | | 1 1 | EMP_NUMBER | CHAR(6), | STR00250 831004 |
| | 1100 | 3.1 | 1 1 | EMP_NAME | CHAR(20), | STR00260 |
| | 1200 | 3.2 | 1 1 | EMP_RATE | DECIMAL FIXED(5,2), | STR00270 831004 |
| | 1300 | 3.3 | 1 1 | EMP_HOURS | DECIMAL FIXED(4,1), | STR00280 831004 |
| | 1400 | 3.4 | 1 1 | EMP_DEDUCTIONS | DECIMAL FIXED(6,2); | STR00290 831004 |
| | 1500 | | | | | STR00300 |
| | 1600 | | | /* PROGRAM FLAGS */ | | STR00310 |
| | 1700 | 4 | 1 1 | DECLARE | | STR00320 |
| | 1800 | | 1 1 | 1 BIT_FLAGS STATIC, | | STR00330 |
| | 1900 | 4.1 | 1 1 | 2 MORE_RECORDS | BIT(1) ALIGNED, | STR00340 |
| | 2000 | 4.2 | 1 1 | 2 NO | BIT(1) ALIGNED INIT('0'B), | STR00350 |
| | 2100 | 4.3 | 1 1 | 2 YES | BIT(1) ALIGNED INIT('1'B); | STR00360 |
| | 2200 | | | | | STR00370 |
| | 2300 | | | /* PROGRAM VARIABLES */ | | STR00380 |
| | 2400 | 5 | 1 1 | DECLARE | | STR00390 |
| | 2500 | | 1 1 | GROSS_PAY | DECIMAL FIXED(7,2), | STR00400 |
| | 2600 | 5.1 | 1 1 | NET_PAY | DECIMAL FIXED(7,2), | STR00410 |
| | 2700 | 5.2 | 1 1 | OVERTIME | DECIMAL FIXED(4,1), | STR00420 831004 |
| | 2800 | 5.3 | 1 1 | PAGE_NUMBER | BINARY FIXED(2); | STR00430 |
| | 2900 | | | | | STR00440 |
| | 3000 | 6 | 1 1 | ON ENDFILE(EMP_FILE) | | STR00450 |
| | 3100 | | 1 1 | MORE_RECORDS = NO; | | STR00460 |
| | 3200 | | | | | STR00470 |
| | 3300 | 7 | 1 1 | ON ENDPAGE (SYSPRINT) | | STR00480 |
| | 3400 | | 1 1 | BEGIN; | | STR00490 |
| | 3500 | 8 | 3 2 | 6 PUT FILE (SYSPRINT) PAGE EDIT('PAGE ',PAGE_NUMBER) | | STR00500 |
| | 3600 | | 3 2 | (X(81),A(6),F(2)); | | STR00510 |
| | 3700 | 9 | 3 2 | PUT FILE (SYSPRINT) SKIP(3) EDIT('EMPLOYEE PAYROLL') | | STR00520 |
| | 3800 | | 3 2 | (X(38),A(16)); | | STR00530 |
| | 3900 | 10 | 3 2 | PUT FILE (SYSPRINT) SKIP(2) EDIT('EMPLY#','EMPLOYEE NAME','RATE', | | STR00540 |
| | 4000 | | 3 2 | 'REG HRS','OVERTME','GROSS PAY','DEDUCTIONS','NET PAY') | | STR00550 |
| | 4100 | | 3 2 | (A(6),X(6),A(13),X(5),A(4),X(2),A(7),X(2),A(7),X(2),A(9), | | STR00560 |
| | 4200 | | 3 2 | X(2),A(10),X(2),A(7)); | | STR00570 |
| | 4300 | 11 | 3 2 | PAGE_NUMBER = PAGE_NUMBER + 1; | | STR00580 |
| | 4400 | 12 | 3 2 | END; /* BEGIN */ | | STR00590 |
| | 4500 | | | | | STR00600 |
| | 4600 | | | /* MAIN PROGRAM */ | | STR00610 |
| | 4700 | 13 | 1 1 | PAGE_NUMBER = 1; | | STR00620 |
| | 4800 | 14 | 1 1 | MORE_RECORDS = YES; | | STR00630 |
| | 4900 | | | | | STR00640 |
| | 5000 | 15 | 1 1 | 7 OPEN | | STR00650 |
| | 5100 | | 1 1 | FILE (EMP_FILE) TITLE('EMPPFILE'); /* INPUT */ | | STR00660 830919 |
| | 5200 | | | 8 | | STR00670 |
| | 5300 | 16 | 1 1 | SIGNAL ENDPAGE (SYSPRINT); | | STR00680 |

Figure 8-17 (Part 1 of 2). Program Using Stream I/O

| 5728PL1 R01 M00 880715 | | PL/I Source Listing | | PLITST/LP1418 | | 11/30/88 14:21:11 | | Page 3 | |
|------------------------|--------|---------------------|-----|---------------|----|--|--|--------|--|
| Include | SEQNBR | STMT.SUBS | BLK | BN | DO | LP1418: PROCEDURE; | *<..+....1....+....2....+....3....+....4....+....5....+....6....+....7.>..+....8 | Date | |
| | 5400 | 17 | 1 | 1 | | GET FILE (EMP_FILE) EDIT(EMP_NUMBER,EMP_NAME,EMP_RATE,EMP_HOURS, | STR00690 | | |
| | 5500 | | 1 | 1 | | EMP_DEDUCTIONS) (A(6),A(20),F(5,2),F(4,1),F(6,2)); | STR00700 | 831004 | |
| | 5600 | | | | | | STR00710 | | |
| | 5700 | 18 | 1 | 1 | | DO WHILE(MORE_RECORDS); | STR00720 | | |
| | 5800 | 19 | 1 | 1 | 1 | GROSS_PAY = 0.0; | STR00730 | | |
| | 5900 | 20 | 1 | 1 | 1 | NET_PAY = 0.0; | STR00740 | | |
| | 6000 | 21 | 1 | 1 | 1 | OVERTIME = 0.0; | STR00750 | | |
| | 6100 | 22 | 1 | 1 | 1 | IF EMP_HOURS > 40.0 THEN | STR00760 | | |
| | 6200 | | 1 | 1 | 1 | DO; | STR00770 | | |
| | 6300 | 23 | 1 | 1 | 2 | OVERTIME = EMP_HOURS - 40.0; | STR00780 | | |
| | 6400 | 24 | 1 | 1 | 2 | EMP_HOURS = 40.0; | STR00790 | | |
| | 6500 | 25 | 1 | 1 | 2 | GROSS_PAY = OVERTIME * (1.5 * EMP_RATE); | STR00800 | | |
| | 6600 | 26 | 1 | 1 | 2 | END; | STR00810 | | |
| | 6700 | 27 | 1 | 1 | 1 | GROSS_PAY = GROSS_PAY + (EMP_HOURS * EMP_RATE); | STR00820 | | |
| | 6800 | 28 | 1 | 1 | 1 | NET_PAY = GROSS_PAY - EMP_DEDUCTIONS; | STR00830 | | |
| | 6900 | 29 | 1 | 1 | 1 | PUT FILE (SYSPRINT) SKIP EDIT(EMP_NUMBER,EMP_NAME,EMP_RATE, | STR00840 | | |
| | 7000 | | 1 | 1 | 1 | EMP_HOURS,OVERTIME,GROSS_PAY,EMP_DEDUCTIONS,NET_PAY) (A(6),X(2), | STR00850 | | |
| | 7100 | | 1 | 1 | 1 | A(20),F(6,2),X(2),F(5,1),X(4),F(5,1),X(4),F(8,2),X(6), | STR00860 | 831004 | |
| | 7200 | | 1 | 1 | 1 | F(6,2),X(2),F(8,2)); | STR00870 | 831004 | |
| | 7300 | 30 | 1 | 1 | 1 | GET FILE (EMP_FILE) SKIP EDIT(EMP_NUMBER,EMP_NAME,EMP_RATE, | STR00880 | | |
| | 7400 | | 1 | 1 | 1 | EMP_HOURS,EMP_DEDUCTIONS) (A(6),A(20),F(5,2),F(4,1),F(6,2)); | STR00890 | 831004 | |
| | 7500 | 31 | 1 | 1 | 1 | END; /* DO WHILE */ | STR00900 | | |
| | 7600 | | | | | | STR00910 | | |
| | 7700 | 32 | 1 | 1 | | CLOSE | STR00920 | | |
| | 7800 | | 1 | 1 | | FILE (EMP_FILE); | STR00930 | | |
| | 7900 | 33 | 1 | 1 | | CLOSE | STR00940 | | |
| | 8000 | | 1 | 1 | | FILE (SYSPRINT); | STR00950 | | |
| | 8100 | | | | | | STR00960 | | |
| | 8200 | 34 | 1 | 1 | | END LP1418; | STR00970 | 830817 | |

Figure 8-17 (Part 2 of 2). Program Using Stream I/O

- 1** STREAM data transmission is used with EMP_FILE.
- 2** EMP_FILE is used for INPUT only. No data is directed to it by the program. STREAM INPUT can be omitted for EMP_FILE, because these are the default attributes for file declarations.
- 3** STREAM data transmission is used with SYSPRINT.
- 4** SYSPRINT is used for OUTPUT only. It does not provide data to the program.
- 5** The PRINT attribute specifies that the first character in each record is an ASA carriage control character.
- 6** SYSPRINT is implicitly opened by the first processing of the PUT statement. Any file that is not explicitly opened is implicitly opened by the first data transmission statement that accesses it. It is, however, a good practice to explicitly open all your files.
- 7** EMP_FILE is explicitly opened. In the file declaration, INPUT has already been specified as an attribute of EMP_FILE. With both stream and record files, the data transmission mode can be specified in the file declaration, or in the OPEN statement, or in both places. If you specify the data transmission mode in both places, you must be sure that you specify the same mode both times.

COMMITMENT CONTROL

- 8 The TITLE option is specified, with the parameter 'EMPPFILE'. Because the library and member names are allowed to default, the first member of file EMPPFILE in the library list is opened.
- 9 A list of variables is written to SYSPRINT. The SKIP option is specified, but no parameter is supplied. The compiler supplies a default value of 1, so one line is skipped before the line is printed.

Commitment Control

Commitment control allows you to ensure that when a commitment boundary is reached, multiple changes to data base files are either made permanent or canceled, as a single operation.

PL/I support for commitment control consists of the following:

- COMMITTABLE option on the ENVIRONMENT file declaration attribute to indicate that a file should be placed under commitment control.
- PLICOMMIT built-in procedure to complete changes to the data base.
- PLIROLLBACK built-in procedure to cancel changes to the data base.

The AS/400 System support for commitment control consists of the following:

- The CL command STRCMTCL (Start Commitment Control) to start commitment control. It indicates that commitment control can be used, and establishes the record locking at either the high level (*ALL) or the low level (*CHG).
 - *ALL record locking is useful primarily to protect concurrent jobs from uncommitted changes.
 - *CHG record locking is useful primarily to simplify recovery.
- The CL command ENDCMTCTL (End Commitment Control) ends commitment control. This command cancels any changes to the data base files that are under commitment control.

For more information on commitment control, see the *Programming: Control Language Programmer's Guide*. For more information on the STRCMTCTL and ENDCMTCTL commands, refer to the *Programming: Control Language Reference*.

PLICOMMIT and PLIROLLBACK are used to maintain a collection of one or more data base files in a consistent state. Changes made to data base files under commitment control are only completed when a PLICOMMIT is processed. Use the PLICOMMIT built-in subroutine to complete or commit all changes made through the WRITE, REWRITE, and DELETE statements, and the PLIROLLBACK built-in subroutine to remove or roll back any changes made by WRITE, REWRITE, and DELETE statements since the last PLICOMMIT or PLIROLLBACK. If a problem occurs during processing, the changes made since the last PLICOMMIT call can be removed by processing PLIROLLBACK.

If the commitment control environment is abnormally ended (either by a system or job failure), the system will ensure that any uncommitted changes are removed from the data base (an implicit rollback operation is processed).

The commit and rollback operations may also be processed by statements in other high-level languages. Commit or rollback operations processed by any program of the job written in any language apply to all committable files that your PL/I program is using. This feature allows the user to keep data base files at consistent transaction boundaries.

Using the COMMITTABLE Option

The format for the commitment control option is:

```

▶▶—ENVIRONMENT(COMMITTABLE)—▶▶
    
```

The COMMITTABLE option and the INTERACTIVE file declaration attribute are mutually exclusive.

A file with the COMMITTABLE option specified is placed in the commitment control environment when the file is opened. The file is then implicitly acted upon by the built-in subroutines PLICOMMIT and PLIROLLBACK.

If the file cannot be placed under commitment control, a message is sent to the job log and the UNDEFINEDFILE condition (93) is raised. The UNDEFINEDFILE condition is also raised. if the file is not a data base file.

If you want one or more different PL/I programs and two different run units, or a run unit and a non-PL/I program to open a file in the commitment control environment, each run unit or non-PL/I program that will open the file must specify the COMMITTABLE option.

Using the PLICOMMIT Built-In Subroutine

The PLICOMMIT built-in subroutine processes commitment control functions.

```

▶▶—CALL—PLICOMMIT—(character_expression)—▶▶;
    
```

character_expression

A character expression that can be converted to a non-varying character variable. A character string of zero length is equivalent to not specifying the argument.

PLICOMMIT processes the commitment control function by establishing a new commitment boundary for the job.

- For files in a commitment control environment, all changes made to these files since the previous commitment boundary are made permanent. Changes are

COMMITMENT CONTROL

made to all files under commitment control in the job, not just to those files in the program that calls PLICOMMIT.

- For files in a commitment control environment, all record locks held by the job since the last commitment boundary are released and the records are made available to other jobs.
- PLICOMMIT only affects files under commitment control. If PLICOMMIT is called and there are no files under commitment control, no functions are processed, and there is no error indication.
- A file under commitment control may be closed or opened without affecting the status of any records that are pending a commit. The file does not have to be open in order to commit records for that file. For example, if since the last commitment boundary, records are updated in an open file and the file is then closed, a call to PLICOMMIT makes the updated records permanent and the file remains closed.
- The end of a procedure or a run unit has no effect on the commitment control environment. Even though files are closed, all uncommitted changes remain pending.
- The end of the job causes an automatic rollback of uncommitted records for all files under commitment control. Any uncommitted changes to the data base are cancelled.
- The character_expression provides up to 2000 bytes of string data to be used as a description (commit id) for the commitment boundary. If the character-expression is not specified (or is length zero) no commitment boundary description is used. For further information, see the description of the NFYOBJ (Notify Object) parameter of the CL command STRCMTCTL in the *Programming: Control Language Reference*.

PLICOMMIT does **not** do any of the following:

- Change the position of a file.
- Raise the ERROR condition.
- Modify the I-O feedback area for a file.
- Change the open or close state of a file.

Using the PLIROLLBACK Built-In Subroutine

PLIROLLBACK processes the rollback function by reestablishing a previous commitment boundary.

```
▶▶CALL PLIROLLBACK;◀◀
```

- PLIROLLBACK removes all changes that have been made to the files since the previous commitment boundary. This applies to all the files in a commitment control environment in the job, and not just those files in the program that called PLIROLLBACK.
- For files in a commitment control environment, all record locks held by the job are released, and the records are made available to other jobs.

- The position of each file under commitment control is set to the previous commitment boundary. If the file was open at the previous commitment boundary and has since been closed, the position of the file is undefined.
- **PLIROLLBACK** only affects files under commitment control. A call to **PLIROLLBACK** is ignored if there are no files under commitment control.
- A file under commitment control may be closed and opened without affecting the status of any records that are pending a rollback. The file does not have to be open in order to rollback records for that file. If, since the last commitment boundary, records are updated and the file is then closed, this procedure removes the updated records and the file remains closed.
- The end of a procedure or a run unit has no effect on the commitment control environment. Even though files are closed, any uncommitted changes remain pending.
- The end of the job causes an automatic rollback of uncommitted records for all files under commitment control. Any uncommitted changes to the data base are cancelled.

PLIROLLBACK does **not** do any of the following:

- Raise the **ERROR** condition.
- Modify the I-O feedback area for a file.
- Change the open or closed state of any file.

Using **PLICOMMIT** and **PLIROLLBACK**

You must call the **PLICOMMIT** and **PLIROLLBACK** procedures at the appropriate times in your *PL/I* program.

Not all of the files that your program uses must be under commitment control. For example, a work file may not need the protection offered by commitment control.

Performance Considerations

Performance may be adversely affected by using commitment control.

- Programs using commitment control may have longer run times.
- Other jobs attempting to access files under commitment control may experience delays.

You can avoid problems if you write programs so that a given file is always run under commitment control or else is never run under commitment control. It is more difficult to design a program so that it could run one time with commitment control and another time without it. This is the case because the file and record lock recovery logic in a program depends upon if the file is under commitment control or not.

Programs using commitment control can be run in multiple jobs concurrently. The system keeps track of the uncommitted data in each job.

COMMITMENT CONTROL

Commitment control is not the only way to provide files with recovery capability. For information on other methods see *Programming: Control Language Programmer's Guide*.

Record Locks

Record locks are specified on the LCKLVL parameter of the CL command STRCMTCTL:

- *CHG specifies that only the records that you change in the files under commitment control are locked until they are committed or rolled back, or until the routing step ends.
- *ALL specifies that any records that are accessed in any files in a commitment control environment are locked until they are committed or rolled back, or until the routing step ends.

For further information on record locks, see the *Programming: Control Language Programmer's Guide*.

Use record locks to prevent changes to data in one job from interfering with data in another job. Because records changed under commitment control are not intended for use by other jobs until a PLICOMMIT or PLIROLLBACK is done, all records changed by a REWRITE, DELETE, or WRITE statement since the previous commitment boundary are locked.

If, in one or more jobs, you use both committed and uncommitted data at the same time, you may find problems if either of the following conditions occurs:

- Records in a physical file are accessed with commitment control and without commitment control at the same time. (For example, if there are two different logical files based on the same physical file.) This is a problem if the two files which access the records are in the same job or in different jobs.
- You specify low level record locking (*CHG) on the CL command STRCMTCTL for a file with INPUT attribute.

When either of these conditions exist, records can be accessed in a data base file (even though no rollback or commit has been done yet) that have the following characteristics:

- Records which have been updated by a REWRITE will contain the updated values.
- Records which have been added by a WRITE will be present.
- Records which have been deleted by a DELETE will not be present.

Note: If an attempt is made to add a record through another access path with the same key as the deleted record, a KEY condition (duplicate key) will exist if duplicates are not allowed.

You may avoid accessing uncommitted records from another job by placing the file under commitment control with a locking level of *ALL. Then, all READ statements on INPUT files attempt to obtain a lock on the record prior to reading the

COMMITMENT CONTROL

record. Because uncommitted records are locked to the job that changed them until the PLICOMMIT or PLIROLLBACK is processed, the job requesting the record will wait until either the record is committed or rolled back, or the wait time ends (WAITRCD).

The following tables show the relationship between the types of locks obtained, lock checking, and the record locking specified by the CL command STRCMTCTL.

| STATEMENT | OPEN OPTION OR FILE ATTRIBUTES | LOCK LEVEL | | CHECK LOCK STATE BEFORE (1) | DURATION OF RECORD LOCK | | | Record Locked After Statement Processed |
|-----------|--------------------------------|------------|---------------|-----------------------------|-------------------------|--------------------|-------------------------------|---|
| | | | | | Lock Maintained | Next I/O Operation | Next PLICOMMIT or PLIROLLBACK | |
| DELETE | UPDATE | BGNCMTCTL | *ALL | Yes | ○ | → | → | Yes |
| | | | *CHG | Yes | ○ | → | → | Yes |
| | | | Not Committed | Yes | ○ | → | → | No |
| READ | UPDATE | BGNCMTCTL | *ALL | Yes | ○ | → | → | Yes |
| | | | *CHG | Yes | ○ | → | → (2) | Yes |
| | | | Not Committed | Yes | ○ | → | → | Yes |
| READ | INPUT | BGNCMTCTL | *ALL | Yes | ○ | → | → | Yes |
| | | | *CHG | No | ○ | → | → | No |
| | | | Not Committed | No | ○ | → | → | No |
| REWRITE | UPDATE | BGNCMTCTL | *ALL | Yes | ○ | → | → | Yes |
| | | | *CHG | Yes | ○ | → | → | Yes |
| | | | Not Committed | Yes | ○ | → | → | No |
| WRITE | UPDATE or OUTPUT | BGNCMTCTL | *ALL | No | ○ | → | → | No |
| | | | *CHG | No | ○ | → | → | Yes |
| | | | Not Committed | No | ○ | → | → | No |

Notes:

1. For DELETE or REWRITE, the lock state is checked before for a KEYED file only, not for a SEQUENTIAL file.
2. If the next operation is a SEQUENTIAL DELETE or REWRITE, the lock state is maintained until the next PLICOMMIT or PLIROLLBACK.

COMMITMENT CONTROL

Error Conditions

The following run time conditions may arise due to problems in a commitment control environment:

| Condition | Statement Involved |
|---------------|--------------------|
| ERROR | DELETE |
| ERROR | REWRITE |
| TRANSMIT | READ |
| UNDEFINEDFILE | OPEN |

Figure 8-18. Run-time Conditions

For more information on the meaning of these conditions, see the discussion of the appropriate condition in "Conditions" on page D-1.

Recovery After a Failure

For a system failure, the files under commitment control are automatically rolled back to the last commitment boundary when the system is restarted.

For a job failure (either because of a user or system error), the files under commitment control are rolled back to the last commitment boundary as part of job end.

Examples Using Commitment Control

You may call a PL/I program in the commitment control environment with the following CL statements:

```
STRCMTCTL LCKLVL(*ALL)  
CALL PGM(LP1425)  
ENDCMTCTL
```

The following example is a program updating a keyed file (see Figure 8-5 on page 8-10) that has been adapted for commitment control.

COMMITMENT CONTROL

```

5728PL1 R01 M00 880715          PL/I Source Listing          PLITST/LP1425          11/30/88 14:49:40          Page 2
LP1425: PROCEDURE;
Include  SEQNBR STMT.SUBS BLK BN DO  *<..+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....>.....8 Date
100      1          1          1          LP1425: PROCEDURE;                                COM00160 830817
200                                             COM00170
300                                             COM00180
400      2          1  1          /* FILE DECLARATIONS */
500                                             DECLARE                                COM00190
600                                             IN_FILE FILE RECORD INTERNAL SEQUENTIAL INPUT ENV(CONSECUTIVE COM00200 831013
700      2.1        1  1          MST_FILE FILE RECORD INTERNAL DIRECT UPDATE ENV(INDEXED COMMITTABLECOM00210 830919
800                                             KEYDISP(0) KEYLENGTH(10)),COM00220 831114
900      2.2        1  1          SYSPRINT FILE STREAM OUTPUT PRINT;                                830607
1000                                             COM00230
1100                                             /* RECORD DECLARATIONS */
1200      3          1  1          DECLARE                                COM00240
1300      1          1  1          1 MASTER_RECORD,                                COM00250
1400      2          1  1          2 MASTER_KEY,                                COM00260 831114
1500      3          1  1          3 MASTER_GEN_FLD CHAR(5),                                COM00280
1600      3          1  1          3 MASTER_DET_FLD CHAR(5),                                COM00290
1700      4          1  1          2 MASTER_NAME CHAR(20),                                COM00300
1800      5          1  1          2 MASTER_BAL PICTURE '999999V9R', COM00310
1900      3.6        1  1          1 INPUT_RECORD,                                COM00320 831013
2000      3.7        1  1          2 INPUT_KEY,                                COM00330
2100      3.8        1  1          3 INPUT_GEN_FLD CHAR(5),                                COM00340
2200      3.9        1  1          3 INPUT_DET_FLD CHAR(5),                                COM00350
2300      3.10       1  1          2 INPUT_NAME CHAR(20),                                COM00360
2400      3.11       1  1          2 INPUT_AMT PICTURE 'S99999V99'; COM00370
2500                                             COM00380 831013
2600                                             COM00390
2700      4          1  1          /* PROGRAM FLAGS */
2800      1          1  1          DECLARE                                COM00400
2900      2          1  1          1 BIT_FLAGS STATIC,                                COM00410
3000      4.1        1  1          2 MORE_RECORDS BIT(1) ALIGNED, COM00420
3100      4.2        1  1          2 NO BIT(1) ALIGNED INIT('0'B), COM00430
3200      4.3        1  1          2 YES BIT(1) ALIGNED INIT('1'B); COM00440
3300                                             COM00450
3400      5          1  1          /* PROGRAM VARIABLES */
3500      1          1  1          DECLARE                                COM00460
3600      5.1        1  1          1 PLICOMMIT BUILTIN,                                COM00470
3700      5.2        1  1          1 PLIROLLBACK BUILTIN,                                COM00480
3800      5.3        1  1          1 OLD_MASTER_BAL PICTURE 'S99999V99', COM00490 830615
3900      1          1  1          1 PAGE_NUMBER BINARY FIXED(2); COM00500 830615
4000      6          1  1          ON ENDFILE(IN_FILE)                                COM00510
4100      1          1  1          MORE_RECORDS = NO;                                COM00520
4200                                             COM00530
4300      7          1  1          ON ENDPAGE (SYSPRINT)                                COM00540
4400      1          1  1          BEGIN;                                COM00550
4500      8          3  2          PUT FILE (SYSPRINT) PAGE EDIT('PAGE ',PAGE_NUMBER) COM00560
4600      3          3  2          (X(81),A(6),F(2));                                COM00570
4700      9          3  2          PUT FILE (SYSPRINT) SKIP(3) EDIT('UPDATE REPORT')(X(38),A(13)); COM00580
4800      10         3  2          PUT FILE (SYSPRINT) SKIP(2) EDIT('KEY ID','NAME','CUR BALANCE', COM00590
4900      3          3  2          'UPDATE AMOUNT','NEW BALANCE')(A(6),X(9),A(4),X(21),A(11), COM00600
5000      3          3  2          X(6),A(13),X(4),A(11));                                COM00610
5100      11         3  2          PAGE_NUMBER = PAGE_NUMBER + 1; COM00620
5200      12         3  2          END; /* BEGIN */                                COM00630
5300                                             COM00640

```

Figure 8-19 (Part 1 of 3). PL/I Program Using Commitment Control

COMMITMENT CONTROL

| 5728PL1 R01 M00 880715 | PL/I Source Listing | PLITST/LP1425 | 11/30/88 14:49:40 | Page 3 |
|------------------------|----------------------------|--|-------------------|-----------------|
| Include | SEQNBR STMT.SUBS BLK BN DO | *<.+....1....+....2....+....3....+....4....+....5....+....6....+....7.>+....8 Date | | |
| | 5400 | LP1425: PROCEDURE; | | COM00160 |
| | 5400 | /* MAIN PROGRAM */ | | COM00680 |
| | 5500 13 1 1 | PAGE_NUMBER = 1; | | COM00690 |
| | 5600 14 1 1 | MORE_RECORDS = YES; | | COM00700 |
| | 5700 | | | COM00710 |
| | 5800 15 1 1 | OPEN | | COM00720 |
| | 5900 1 1 | FILE (IN_FILE) TITLE('UPDATES'); /* INPUT */ | | COM00730 831013 |
| | 6000 16 1 1 | OPEN | | COM00740 |
| | 6100 1 1 | FILE (MST_FILE) TITLE('MSTFILE'); /* UPDATE */ | | COM00750 830919 |
| | 6200 | | | COM00760 |
| | 6300 17 1 1 | SIGNAL ENDPAGE (SYSPRINT); | | COM00770 |
| | 6400 18 1 1 | READ FILE (IN_FILE) INTO (INPUT_RECORD); | | COM00780 |
| | 6500 | | | COM00790 |
| | 6600 19 1 1 | DO WHILE(MORE_RECORDS); | | COM00800 |
| | 6700 20 1 1 1 | IF INPUT_DET_FLD = ' ' THEN | | COM00810 |
| | 6800 1 1 1 | CALL INITSEQ; | | COM00820 |
| | 6900 21 1 1 1 | ELSE | | COM00830 |
| | 7000 1 1 1 | CALL DYNAMIC; | | COM00840 |
| | 7100 22 1 1 1 | READ FILE (IN_FILE) INTO (INPUT_RECORD); | | COM00850 |
| | 7200 23 1 1 1 | END; /* DO WHILE */ | | COM00860 |
| | 7300 | | | COM00870 |
| | 7400 24 1 1 | CLOSE | | COM00880 |
| | 7500 1 1 | FILE (IN_FILE); | | COM00890 |
| | 7600 25 1 1 | CLOSE | | COM00900 |
| | 7700 1 1 | FILE (MST_FILE); | | COM00910 |
| | 7800 | | | COM00920 |
| | 7900 26 1 1 | INITSEQ: PROCEDURE; | | COM00930 |
| | 8000 27 4 2 | MASTER_GEN_FLD = INPUT_GEN_FLD; | | COM00940 |
| | 8100 28 4 2 | READ FILE (MST_FILE) INTO (MASTER_RECORD) KEY(MASTER_KEY) | | COM00950 831013 |
| | 8200 4 2 | OPTIONS(KEYSEARCH(EQLAFT) NBRKEYFLDS(1)); | | COM00960 831013 |
| | 8300 29 4 2 | DO WHILE(INPUT_GEN_FLD = MASTER_GEN_FLD); | | COM00970 |
| | 8400 30 4 2 1 | CALL SEQPROC; | | COM00980 |
| | 8500 31 4 2 1 | END; /* DO WHILE */ | | COM00990 |
| | 8600 32 4 2 | RETURN; | | COM01000 |
| | 8700 33 4 2 | END INITSEQ; | | COM01010 |
| | 8800 | | | COM01020 |
| | 8900 34 1 1 | SEQPROC: PROCEDURE; | | COM01030 |
| | 9000 35 5 2 | PUT FILE (SYSPRINT) SKIP EDIT(MASTER_KEY,MASTER_NAME, | | COM01070 831013 |
| | 9100 5 2 | MASTER_BAL)(A(5),A(5),X(5),A(20),X(6),F(10,2)); | | COM01080 831013 |
| | 9200 36 5 2 | READ FILE (MST_FILE) INTO (MASTER_RECORD) KEY(MASTER_KEY) | | COM01040 831013 |
| | 9300 5 2 | OPTIONS(KEYSEARCH(AFTER)); | | COM01050 831013 |
| | 9400 37 5 2 | RETURN; | | COM01090 |
| | 9500 38 5 2 | END SEQPROC; | | COM01100 |
| | 9600 | | | COM01110 |
| | 9700 39 1 1 | DYNAMIC: PROCEDURE; | | COM01120 |
| | 9800 40 6 2 | MASTER_KEY = INPUT_KEY; | | COM01130 |
| | 9900 41 6 2 | READ FILE (MST_FILE) INTO (MASTER_RECORD) KEY(MASTER_KEY) | | COM01140 831013 |
| | 10000 6 2 | OPTIONS(KEYSEARCH(EQUAL)); | | 831013 |
| | 10100 42 6 2 | IF INPUT_GEN_FLD = MASTER_GEN_FLD THEN | | COM01150 |
| | 10200 6 2 | DO; | | COM01160 |
| | 10300 43 6 2 1 | OLD_MASTER_BAL = MASTER_BAL; | | COM01170 |
| | 10400 44 6 2 1 | MASTER_BAL = MASTER_BAL + INPUT_AMT; | | COM01180 |
| | 10500 45 6 2 1 | PUT FILE (SYSPRINT) SKIP EDIT(MASTER_KEY,MASTER_NAME, | | COM01190 |
| | 10600 6 2 1 | OLD_MASTER_BAL,INPUT_AMT,MASTER_BAL)(A(5),A(5),X(5),A(20), | | COM01200 831013 |

Figure 8-19 (Part 2 of 3). PL/I Program Using Commitment Control

| 5728PL1 R01 M00 880715 | | PL/I Source Listing | | PLITST/LP1425 | | 11/30/88 14:49:40 | | Page 4 | |
|------------------------|--------|---------------------|-----|---------------|----|--|---|----------|--------|
| Include | SEQNBR | STMT.SUBS | BLK | BN | DO | *<.....1.....2.....3.....4.....5.....6.....7.....>.....8 | Date | | |
| | 10700 | | 6 | 2 | 1 | | | COM00160 | |
| | 10800 | 46 | 6 | 2 | 1 | | X(6),F(10,2),X(6),F(10,2),X(8),F(10,2)); | COM01210 | 831013 |
| | 10900 | | 6 | 2 | 1 | | REWRITE FILE (MST_FILE) FROM (MASTER_RECORD) KEY(MASTER_KEY); | COM01220 | |
| | 11000 | 47 | 6 | 2 | 1 | 4 | IF MASTER_BAL >= 0.0 THEN | COM01230 | |
| | 11100 | | 6 | 2 | 1 | | DO; | COM01240 | |
| | 11200 | 48 | 6 | 2 | 2 | 5 | CALL PLICOMMIT; | COM01250 | |
| | 11300 | 49 | 6 | 2 | 2 | | PUT FILE (SYSPRINT) EDIT('* TRANSACTION COMMITTED *') | COM01260 | 830615 |
| | 11400 | | 6 | 2 | 2 | | (X(1),A(26)); | COM01270 | 831013 |
| | 11500 | 50 | 6 | 2 | 2 | | END; /* DO */ | COM01280 | 831013 |
| | 11600 | 51 | 6 | 2 | 1 | | ELSE | COM01290 | |
| | 11700 | | 6 | 2 | 1 | | DO; | COM01300 | |
| | 11800 | 52 | 6 | 2 | 2 | 6 | CALL PLIROLLBACK; | COM01310 | |
| | 11900 | 53 | 6 | 2 | 2 | | PUT FILE (SYSPRINT) EDIT('* TRANSACTION ROLLEDBACK *') | COM01320 | 830615 |
| | 12000 | | 6 | 2 | 2 | | (X(1),A(26)); | COM01330 | 831013 |
| | 12100 | 54 | 6 | 2 | 2 | | END; /* DO */ | COM01340 | 831013 |
| | 12200 | | | | | | | COM01350 | |
| | 12300 | 55 | 6 | 2 | 1 | | END; /* DO */ | COM01360 | |
| | 12400 | 56 | 6 | 2 | | | RETURN; | COM01370 | |
| | 12500 | 57 | 6 | 2 | | | END DYNAMIC; | COM01380 | |
| | 12600 | | | | | | | COM01390 | |
| | 12700 | 58 | 1 | 1 | | | END LP1425; | COM01400 | |
| | | | | | | | | COM01410 | 830817 |

Figure 8-19 (Part 3 of 3). PL/I Program Using Commitment Control

- 1 See Figure 8-5 on page 8-10 for an explanation of this same program without commitment control.
- 2 Code the COMMITTABLE option in the ENVIRONMENT attribute.
- 3 Declare the built-in subroutines PLICOMMIT and PLIROLLBACK.
- 4 Check for successful transactions.
- 5 Commit the transaction,
- 6 Or roll back the transaction.

The following example uses a display file and indicators to check for a successful transaction in a program to update a file.

COMMITMENT CONTROL

```
5728PL1 M01 M00 880715          PL/I Source Listing          LP1426/LP1426          11/30/88 15:41:06          Page 2
LP1426: PROCEDURE;
LP1426: PROCEDURE;
LP1426: PROCEDURE;

/* FILE DECLARATIONS */
DECLARE
  ACT_FILE FILE RECORD INTERNAL DIRECT UPDATE ENV(INDEXED COMMITTABLECCT00200 830919
  KEYDISP(0) KEYLENGTH(5)),CCT00210 830919
  ACCTFMTS FILE RECORD INTERNAL SEQUENTIAL UPDATE ENV(INTERACTIVE); CCT00220 830607

/* RECORD DECLARATIONS */
DECLARE
  1 FROM_ACCOUNT_RECORD,
    2 ACCT_KEY          PICTURE '9999R',          CCT00270 831013
    2 NAME              CHAR(20),                 CCT00280
    2 ADDR              CHAR(20),                 CCT00290
    2 CITY              CHAR(20),                 CCT00300
    2 STATE             CHAR(2),                  CCT00310
    2 ZIP               PICTURE '99999R',         CCT00320 831013
    2 BALANCE           PICTURE '9999999V9R',     CCT00330 831013
  1 TO_ACCOUNT_RECORD,
    2 ACCT_KEY          PICTURE '9999R',          CCT00350 831013
    2 NAME              CHAR(20),                 CCT00360
    2 ADDR              CHAR(20),                 CCT00370
    2 CITY              CHAR(20),                 CCT00380
    2 STATE             CHAR(2),                  CCT00390
    2 ZIP               PICTURE '99999R',         CCT00400 831013
    2 BALANCE           PICTURE '9999999V9R';     CCT00410 831013

DECLARE
  1 DISPLAY_RECORD,
    %INCLUDE ACCTFMTS(ACCTPMT,INPUT,,COMMA);
    CCT00420
    CCT00430 831013

ACCTPMT + 100  /* ----- */
ACCTPMT + 200  /* DEVICE FILE: ACCTFMTS.LP1426 */
ACCTPMT + 300  /* FILE CREATION DATE: 87/11/30 */
ACCTPMT + 400  /* RECORD FORMAT: ACCTPMT */
ACCTPMT + 500  /* RECORD FORMAT SEQUENCE ID: 11FB7D353213F */
ACCTPMT + 600  /* ----- */
ACCTPMT + 700  /* INDICATORS FOR FORMAT ACCTPMT */
ACCTPMT + 800  /* INDICATOR 15      END OF PROGRAM */
ACCTPMT + 900  /* INDICATOR 94     INVALID TRANSACTION AMOUNT INPUTTED */
ACCTPMT + 1000 /* INDICATOR 97     INVALID TO ACCOUNT NUMBER */
ACCTPMT + 1100 /* INDICATOR 98     INSUFFICIENT FUNDS IN FROM ACCOUNT */
ACCTPMT + 1200 /* INDICATOR 99     INVALID FROM ACCOUNT NUMBER */
ACCTPMT + 1300 /* -----CUSTOMER ACCOUNT PROMPT----- */
ACCTPMT + 1400 15 ACCTFROM PIC '9999R',
ACCTPMT + 1500 15 ACCTTO  PIC '9999R',
ACCTPMT + 1600 15 TRANSAMT PIC '9999999V9R',
3100 4.4 1 1 1 1
3200 1 DISPLAY_INDICATORS,
    %INCLUDE ACCTFMTS(ACCTPMT,INDICATORS,,COMMA);
    CCT00440
    CCT00450 830919

ACCTPMT + 100  /* ----- */
ACCTPMT + 200  /* DEVICE FILE: ACCTFMTS.LP1426 */
ACCTPMT + 300  /* FILE CREATION DATE: 87/11/30 */
ACCTPMT + 400  /* RECORD FORMAT: ACCTPMT */
ACCTPMT + 500  /* RECORD FORMAT SEQUENCE ID: 11FB7D353213F */

Include   SEQNBR STMT.SUBS BLK BN DO
100      1
200
300
400      2      1 1
500      1 1
600      1 1
700      2.1    1 1
800
900
1000     3      1 1
1100     1 1
1200     3.1    1 1
1300     3.2    1 1
1400     3.3    1 1
1500     3.4    1 1
1600     3.5    1 1
1700     3.6    1 1
1800     3.7    1 1
1900     3.8    1 1
2000     3.9    1 1
2100     3.10   1 1
2200     3.11   1 1
2300     3.12   1 1
2400     3.13   1 1
2500     3.14   1 1
2600     3.15   1 1
2700
2800     4      1 1
2900     1 1
3000
ACCTPMT + 100
ACCTPMT + 200
ACCTPMT + 300
ACCTPMT + 400
ACCTPMT + 500
ACCTPMT + 600
ACCTPMT + 700
ACCTPMT + 800
ACCTPMT + 900
ACCTPMT + 1000
ACCTPMT + 1100
ACCTPMT + 1200
ACCTPMT + 1300
ACCTPMT + 1400 4.1 1 1
ACCTPMT + 1500 4.2 1 1
ACCTPMT + 1600 4.3 1 1
3100 4.4 1 1
3200
ACCTPMT + 100
ACCTPMT + 200
ACCTPMT + 300
ACCTPMT + 400
ACCTPMT + 500
```

Figure 8-20 (Part 3 of 6). Program Using PLICOMMIT and PLIROLLBACK

COMMITMENT CONTROL

```

5728PL1 R01 M00 880715          PL/I Source Listing          LP1426/LP1426          11/30/88 15:41:06          Page 3
                                LP1426: PROCEDURE;          CCT00160
Include      SEQNBR STMT.SUBS BLK BN DO  *<.....1.....2.....3.....4.....5.....6.....7.....8 Date
ACCTPMT     + 600                                /* -----CUSTOMER ACCOUNT PROMPT----- */
ACCTPMT     + 700      4.5      1 1      15 IN01_IN14 CHAR(14),      /* UNDEFINED INDICATOR(S) */
ACCTPMT     + 800      4.6      1 1      15 IN15      PIC '9',      /* END OF PROGRAM */
ACCTPMT     + 900      4.7      1 1      15 IN16_IN93 CHAR(78),      /* UNDEFINED INDICATOR(S) */
ACCTPMT     + 1000     4.8      1 1      15 IN94      PIC '9',      /* INVALID TRANSACTION AMOUNT
ACCTPMT     + 1100                                INPUTTED */
ACCTPMT     + 1200     4.9      1 1      15 IN95_IN96 CHAR(02),      /* UNDEFINED INDICATOR(S) */
ACCTPMT     + 1300     4.10     1 1      15 IN97      PIC '9',      /* INVALID TO ACCOUNT NUMBER */
ACCTPMT     + 1400     4.11     1 1      15 IN98      PIC '9',      /* INSUFFICIENT FUNDS IN FROM
ACCTPMT     + 1500                                ACCOUNT */
ACCTPMT     + 1600     4.12     1 1      15 IN99      PIC '9',      /* INVALID FROM ACCOUNT NUMBER */
                                1 ERROR FORMAT,      CCT00460
                                %INCLUDE ACCTFMTS(ERRFMT,OUTPUT,,COMMA);      CCT00470 831013
ERRFMT      + 100                                /* ----- */
ERRFMT      + 200                                /* DEVICE FILE: ACCTFMTS.LP1426 */
ERRFMT      + 300                                /* FILE CREATION DATE: 87/11/30 */
ERRFMT      + 400                                /* RECORD FORMAT: ERRFMT */
ERRFMT      + 500                                /* RECORD FORMAT SEQUENCE ID: 0061A19C6D524 */
ERRFMT      + 600                                /* ----- */
ERRFMT      + 700                                /* INDICATORS FOR FORMAT ERRFMT */
ERRFMT      + 800                                /* INDICATOR 95 */
ERRFMT      + 900                                /* INDICATOR 96 */
ERRFMT      + 1000                               /* ----- */
ERRFMT      + 1100     4.14     1 1      15 DUMMYDCL CHAR(0),      /* NO FIELDS OF NEEDED TYPE */
                                1 ERROR_FORMAT_INDICATORS,      CCT00480
                                %INCLUDE ACCTFMTS(ERRFMT,INDICATORS);      CCT00490 830607
ERRFMT      + 100                                /* ----- */
ERRFMT      + 200                                /* DEVICE FILE: ACCTFMTS.LP1426 */
ERRFMT      + 300                                /* FILE CREATION DATE: 87/11/30 */
ERRFMT      + 400                                /* RECORD FORMAT: ERRFMT */
ERRFMT      + 500                                /* RECORD FORMAT SEQUENCE ID: 0061A19C6D524 */
ERRFMT      + 600                                /* ----- */
ERRFMT      + 700      4.16     1 1      15 IN01_IN94 CHAR(94),      /* UNDEFINED INDICATOR(S) */
ERRFMT      + 800      4.17     1 1      15 IN95      PIC '9',
ERRFMT      + 900      4.18     1 1      15 IN96      PIC '9',
ERRFMT      + 1000     4.19     1 1      15 IN97_IN99 CHAR(03);      /* UNDEFINED INDICATOR(S) */
                                CCT00500
                                /* INDICATOR FLAGS */
                                CCT00510
                                DECLARE      CCT00520
                                1 INDICATOR_FLAGS STATIC,      CCT00530
                                2 OFF      PICTURE '9' INIT(0),      CCT00540 831013
                                2 ON      PICTURE '9' INIT(1);      CCT00550 831013
                                CCT00560
                                /* BUILT-IN FUNCTIONS */
                                CCT00570
                                DECLARE      CCT00580
                                ONCODE      BUILTIN,      CCT00590
                                2 PLICOMMIT      BUILTIN,      CCT00600 830919
                                2 PLIROLLBACK      BUILTIN;      CCT00610 830919
                                CCT00620
                                /* PROGRAM VARIABLES */
                                CCT00630
                                DECLARE      CCT00640
                                OPERATION      CHAR(9);      CCT00650
                                CCT00660

```

Figure 8-20 (Part 4 of 6). Program Using PLICOMMIT and PLIROLLBACK

COMMITMENT CONTROL

| 5728PL1 R01 M00 880715 | PL/I Source Listing | LP1426/LP1426 | 11/30/88 15:41:06 | Page 4 |
|------------------------|----------------------------|--|-------------------|--------|
| Include | SEQNBR STMT.SUBS BLK BN DO | *<...1...2...3...4...5...6...7...>...8 Date | | |
| | 5400 8 1 1 | LP1426: PROCEDURE; ON KEY (ACT_FILE) | CCT00160 | |
| | 5500 1 1 | BEGIN; | CCT00670 | |
| | 5600 9 2 2 | ON ERROR SYSTEM; | CCT00680 | |
| | 5700 10 2 2 | IF ONCODE = 51 THEN | CCT00690 | |
| | 5800 2 2 | IN96 = ON; | CCT00700 | |
| | 5900 11 2 2 | ELSE | CCT00710 | |
| | 6000 2 2 | IF OPERATION = 'REWRITE' THEN | CCT00720 | |
| | 6100 2 2 | IN95 = ON; | CCT00730 | |
| | 6200 12 2 2 | ELSE | CCT00740 | |
| | 6300 2 2 | IF OPERATION = 'READ-FROM' THEN | CCT00750 | |
| | 6400 2 2 | DO; | CCT00760 | |
| | 6500 13 2 2 1 | IN99 = ON; | CCT00770 | |
| | 6600 14 2 2 1 | GO TO LAB1; | CCT00780 | |
| | 6700 15 2 2 1 | END; /* DO */ | CCT00790 | |
| | 6800 16 2 2 | ELSE | CCT00800 | |
| | 6900 2 2 | IF OPERATION = 'READ-TO' THEN | CCT00810 | |
| | 7000 2 2 | DO; | CCT00820 | |
| | 7100 17 2 2 1 | IN97 = ON; | CCT00830 | |
| | 7200 18 2 2 1 | GO TO LAB2; | CCT00840 | |
| | 7300 19 2 2 1 | END; /* DO */ | CCT00850 | |
| | 7400 20 2 2 | WRITE FILE (ACCTFMTS) FROM (ERROR_FORMAT) | CCT00860 | 830607 |
| | 7500 2 2 | OPTIONS(RECORD('ERRFMT') INDICATORS(ERROR_FORMAT_INDICATORS)); | CCT00870 | |
| | 7600 21 2 2 | CLOSE | CCT00880 | |
| | 7700 2 2 | FILE (ACT_FILE); | CCT00890 | |
| | 7800 22 2 2 | CLOSE | CCT00900 | |
| | 7900 2 2 | FILE (ACCTFMTS); | CCT00910 | |
| | 8000 23 2 2 | STOP; | CCT00920 | 830607 |
| | 8100 24 2 2 | END; /* BEGIN */ | CCT00930 | |
| | 8200 | | CCT00940 | |
| | 8300 | /* MAIN PROGRAM */ | CCT00950 | |
| | 8400 25 1 1 | OPEN | CCT00960 | |
| | 8500 1 1 | FILE (ACT_FILE) TITLE('ACTFILE'); /* UPDATE */ | CCT00970 | |
| | 8600 26 1 1 | OPEN | CCT00980 | 830919 |
| | 8700 1 1 | FILE (ACCTFMTS); /* UPDATE */ | CCT00990 | |
| | 8800 | | CCT01000 | 830919 |
| | 8900 27 1 1 | DISPLAY_INDICATORS = OFF; /* IN15,IN94,IN97,IN98,IN99 */ | CCT01010 | |
| | 9000 28 1 1 | ERROR_FORMAT_INDICATORS = OFF; /* IN95,IN96 */ | CCT01020 | 831102 |
| | 9100 | | CCT01030 | |
| | 9200 | /* DISPLAY THE SCREEN */ | CCT01040 | |
| | 9300 29 1 1 | CALL DISPLAY; | CCT01050 | |
| | 9400 | | CCT01060 | |
| | 9500 30 1 1 | DO WHILE(IN15 = OFF); | CCT01070 | |
| | 9600 | /* CHECK FOR INVALID TRANSACTION AMOUNT */ | CCT01080 | |
| | 9700 31 1 1 1 | IF TRANSAMT < 0.0 THEN | | 831102 |
| | 9800 1 1 1 | DO; | | 831102 |
| | 9900 32 1 1 2 | IN94 = ON; | | 831102 |
| | 10000 33 1 1 2 | GOTO LAB3; | | 831102 |
| | 10100 34 1 1 2 | END; | | 831102 |
| | 10200 | /* VERIFY FROM-ACCOUNT */ | CCT01090 | |
| | 10300 35 1 1 1 | FROM_ACCOUNT_RECORD.ACCT_KEY = ACCTFROM; | CCT01100 | |
| | 10400 36 1 1 1 | OPERATION = 'READ-FROM'; | CCT01110 | |
| | 10500 37 1 1 1 | READ FILE (ACT_FILE) INTO (FROM_ACCOUNT_RECORD) | CCT01120 | |
| | 10600 1 1 1 | KEY(FROM_ACCOUNT_RECORD.ACCT_KEY); | CCT01130 | |

Figure 8-20 (Part 5 of 6). Program Using PLICOMMIT and PLIROLLBACK

COMMITMENT CONTROL

| 5728PL1 R01 M00 880715 | | PL/I Source Listing | | LP1426/LP1426 | 11/30/88 15:41:06 | Page 5 |
|------------------------|--------|---------------------|-----|---------------|-------------------|-----------------|
| Include | SEQNBR | STMT.SUBS | BLK | BN | DO | Date |
| | 10700 | | | | | CCT00160 |
| | 10800 | 38 | 1 | 1 | 1 | CCT01140 |
| | 10900 | | 1 | 1 | 1 | CCT01150 |
| | 11000 | | | | | CCT01160 |
| | 11100 | 39 | 1 | 1 | 2 | CCT01170 |
| | 11200 | 40 | 1 | 1 | 2 | CCT01180 |
| | 11300 | 41 | 1 | 1 | 2 | CCT01190 |
| | 11400 | | 1 | 1 | 2 | CCT01200 |
| | 11500 | | | | | CCT01210 |
| | 11600 | 42 | 1 | 1 | 2 | CCT01220 |
| | 11700 | | 1 | 1 | 2 | CCT01230 |
| | 11800 | 43 | 1 | 1 | 2 | CCT01240 830919 |
| | 11900 | | 1 | 1 | 2 | CCT01250 |
| | 12000 | | | | | CCT01260 |
| | 12100 | 44 | 1 | 1 | 3 | CCT01270 |
| | 12200 | | 1 | 1 | 3 | CCT01280 |
| | 12300 | 45 | 1 | 1 | 3 | CCT01290 |
| | 12400 | | 1 | 1 | 3 | CCT01300 |
| | 12500 | 46 | 1 | 1 | 3 | CCT01310 |
| | 12600 | 47 | 1 | 1 | 3 | CCT01320 |
| | 12700 | | 1 | 1 | 3 | CCT01330 |
| | 12800 | 48 | 1 | 1 | 3 | CCT01340 |
| | 12900 | | 1 | 1 | 3 | CCT01350 |
| | 13000 | 49 | 1 | 1 | 3 | CCT01360 |
| | 13100 | | 1 | 1 | 3 | CCT01370 |
| | 13200 | 50 | 1 | 1 | 4 | 831102 |
| | 13300 | 51 | 1 | 1 | 4 | 830919 |
| | 13400 | 52 | 1 | 1 | 4 | 831102 |
| | 13500 | 53 | 1 | 1 | 3 | 831102 |
| | 13600 | | 1 | 1 | 3 | CCT01390 |
| | 13700 | 54 | 1 | 1 | 3 | CCT01400 830919 |
| | 13800 | 55 | 1 | 1 | 2 | CCT01410 |
| | 13900 | 56 | 1 | 1 | 1 | CCT01420 |
| | 14000 | | 1 | 1 | 1 | 831102 |
| | 14100 | 57 | 1 | 1 | 1 | CCT01430 831102 |
| | 14200 | | | | | CCT01440 |
| | 14300 | 58 | 1 | 1 | | CCT01450 |
| | 14400 | | 1 | 1 | | CCT01460 |
| | 14500 | 59 | 1 | 1 | | CCT01470 |
| | 14600 | | 1 | 1 | | CCT01480 |
| | 14700 | | | | | CCT01490 830607 |
| | 14800 | 60 | 1 | 1 | | CCT01500 |
| | 14900 | | | | | CCT01510 |
| | 15000 | 61 | 3 | 2 | | CCT01520 |
| | 15100 | | 3 | 2 | | CCT01530 830607 |
| | 15200 | 62 | 3 | 2 | | CCT01540 |
| | 15300 | 63 | 3 | 2 | | CCT01550 831102 |
| | 15400 | | 3 | 2 | | CCT01560 830607 |
| | 15500 | 64 | 3 | 2 | | CCT01570 |
| | 15600 | | | | | CCT01580 |
| | 15700 | 65 | 1 | 1 | | CCT01590 |
| | | | | | | CCT01600 830817 |

Figure 8-20 (Part 6 of 6). Program Using PLICOMMIT and PLIROLLBACK

```

ACCOUNT MASTER UPDATE

FROM ACCOUNT NUMBER 1111
TO ACCOUNT NUMBER 12345
AMOUNT TRANSFERRED 5._____

      MAKE SURE THAT YOU INSERT A DECIMAL POINT
      EVEN IF THERE ARE NO CENTS INVOLVED.

      DO NOT PLACE ANY TYPE OF SYMBOL IN FRONT OF OR BEHIND
      THE NUMBER (E.G. $, PLUS SIGN, MINUS SIGN, ETC.).
    
```

Figure 8-21. Display for Program Using PLICOMMIT and PLIROLLBACK

- 1** Code the COMMITTABLE option in the ENVIRONMENT attribute.
- 2** Declare the built-in subroutines PLICOMMIT and PLIROLLBACK.
- 3** Check for successful transactions and:
- 4** Either roll back the transaction,
- 5** Or commit the transaction.

Using the %INCLUDE Directive for External File Descriptions

The %INCLUDE directive can be used to copy external text into the source program, and to copy data description specifications (DDS) for externally described files into the source program.

The %INCLUDE directive has a different format for each of the following functions:

- Copying source text into a source program.

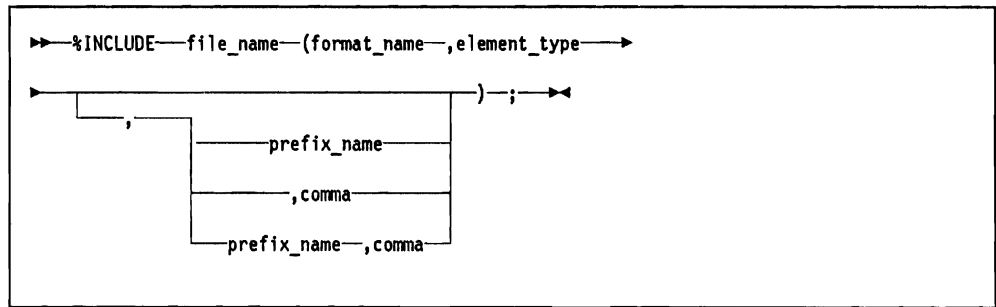
The syntax diagram and description of this function can be found in “Using the %INCLUDE Directive” on page 2-16.

- Copying record formats from AS/400 files into PL/I programs.

This function of the %INCLUDE directive is discussed below.

The syntax of the %INCLUDE directive used for copying record formats from AS/400 files into PL/I programs is shown below:

USING THE %INCLUDE DIRECTIVE



file_name

An identifier of up to 10 characters. The file is located by using the *LIBL search list in effect at compile time. The file name can begin with and contain numeric characters and periods and can use all the characters allowed in a System/38 name. You cannot name your file SYSLIB.

format_name

Name of the record format included. The name must follow the rules for naming AS/400 objects.

element_type

Specifies the fields and indicators that are included. The possible types are:

INPUT

Generates the record definition that matches the input buffer for data base or device files. Includes fields that have a usage of INPUT or BOTH in DDS. For subfiles, output fields are also included. The response indicators are also included when the DDS keyword INDARA is not specified in the external file description.

OUTPUT

Generates the record definition that matches the output buffer for data base files only. Includes fields that have a usage of OUTPUT or BOTH in DDS. The option indicators are also included when the DDS keyword INDARA is not specified in the external file description.

KEY

Includes fields that are specified as keys in DDS.

INDICATORS

Defines a 99-byte area for indicators.

When you specify INDICATORS, you must also specify the following:

- The DDS keyword INDARA in the external description of the file.
- The INDICATORS parameter on the OPTIONS option of any record I/O statements for the file.

RECORD

Generates the record definition that matches both the input and the output buffers that are used by physical and logical files. Includes fields that have a usage of BOTH in DDS. INPUT fields in logical files are also included. It includes any indicators that are used as option indicators and also as response indicators if INDARA is not specified in the external file description.

prefix_name

A character string by which all generated names are prefixed. The prefix is limited to 30 characters or less. The resulting names must be valid and the length must be 31 characters or less.

COMMA

Specifies that the last data element of the record or key structure is followed by a comma. If COMMA is not specified, the last data element is followed by a semicolon. The COMMA option may be used to position the data elements generated by the %INCLUDE directive within a structure without ending the structure.

Using the %INCLUDE Directive with Externally Described Files

When you explicitly declare a file in your program with the DESCRIBED option of the ENVIRONMENT attribute, the following checks are processed for each record format copied into the program from the file with a %INCLUDE directive:

- Level checking, at open time, for each record format. For more information on level checking, see the *Programming: Control Language Programmer's Guide*. To prevent level checking, you may specify no level checking when you create the file, or you may override the default level checking after the file is created (see the CL command CRTxxx F (Create xxx File) and OVRxxx F (Override xxx File) in the *Programming: Control Language Reference*).
- The compatibility of the program description of the file (ENVIRONMENT options) and the external description of the file (DDS) (see Figure 7-2 on page 7-9).

If the externally defined file

- Has no fields defined for it, or
- The element type is INPUT and there exists no fields in the record format with usage INPUT or BOTH, or
- The element type is OUTPUT, the file is not a data base file and there exists no fields in the record format with usage OUTPUT or BOTH, or
- The element type is INDICATORS but no separate indicators exist

then the compiler will generate the following declaration:

```
15 DUMMYDCL CHAR(0);
```

and will produce a comment in the listing and issue a severity 10 warning message. See Figure 8-23 on page 8-79.

For example, a DUMMYDCL statement is generated by a subfile control record format (which has no record fields and exists to define indicators and communicate with the system).

Even if no fields are included and a DUMMYDCL is generated, the compiler processes level checking and compatibility checking.

If you specify DESCRIBED but do not use the %INCLUDE directive in your program, no level checking is processed. This is not the normal case but is accept-

USING THE %INCLUDE DIRECTIVE

able because the DESCRIBED attribute may be used to obtain functions other than level checking.

Using the %INCLUDE Directive with Program-Described Files

If DESCRIBED is not specified, the %INCLUDE can be used to generate record format definitions, but no level checking occurs when the file is opened.

Using the %INCLUDE Directive with Display Files

Input and Output

The INPUT and OUTPUT options are provided for display files. The INPUT option generates fields that have a DDS field usage of INPUT or BOTH, giving a record definition that matches the input buffer. The OUTPUT option generates fields that have a DDS field usage of OUTPUT or BOTH, giving a record definition that matches the output buffer.

If you are using a single record format for both input and output, you should use the %INCLUDE directive twice, specifying element-type INPUT one time and element-type OUTPUT the other time. To avoid duplicating the definition of the record format, you may do one of the following:

- Use a unique prefix
- Include each use of INPUT and OUTPUT in a unique structure.

For example:

```
DECLARE
  1 DISPLAY_SCREEN,
  5 INPUT_FIELDS,
  %INCLUDE FLDREFFILE(INFIELDS,INPUT,,COMMA);
  5 OUTPUT_FIELDS,
  %INCLUDE FLDREFFILE(OUTFIELDS,OUTPUT);
```

If you specify the PL/I type INPUT, either INPUT or BOTH fields should exist in the DDS description of the record format. If you specify the PL/I type OUTPUT, either OUTPUT or BOTH fields should exist in the DDS description of the record format. In either case, if no such fields exists in the record format, the compiler will generate a DUMMYDCL.

Indicators

You can define the indicators for a record format as part of a separate area, independent of the record format, or you can define them as fields in the record format. For a discussion of these methods for using indicators, see the sections below, and refer to "INDICATORS Parameter" on page 7-19.

Indicators in a Separate Area: If you are using indicators in a separate area, make sure that you have done all of the following:

- Specify INDICATORS type on the %INCLUDE directive.
- Specify the DDS keyword INDARA on the external description of the file.

- Specify INDICATORS on the OPTIONS option of any input/output statement that accesses the record format.

Indicators defined for a record format, both at the file level and at the record level, are returned as comments in the header description for each %INCLUDE directive when INPUT, OUTPUT, RECORD, or KEY is specified, if indicators are in a separate indicator area.

Indicators as fields in the record format: Indicators in the buffer are declared as fields in the record when INPUT, OUTPUT, or RECORD is specified.

When the indicators are defined in the record buffer, names are generated in the form of INnn where nn is the number of the indicator for each indicator that is used.

DDS to PL/I Mapping

The following table shows how each DDS data type is defined in a PL/I program:

| DDS Data Type | Length | Decimal Position | Generated PL/I Declaration | Supported by PL/I |
|---------------|------------|------------------|---|-------------------|
| indicator | 1 | 0 | PICTURE '9' | Yes |
| A | 1 - 32 766 | none | CHARACTER (n) where n = 1 to 32 766 | Yes |
| B | 1 - 4 | 0 | BINARY FIXED (15) UNALIGNED | Yes |
| B | 5 - 9 | 0 | BINARY FIXED (31) UNALIGNED | Yes |
| B | 1 - 4 | 1 - 4 | CHARACTER (2) | No |
| B | 5 - 9 | 1 - 9 | CHARACTER (4) | No |
| P | 1 - 15 | 0 - 15 | DECIMAL FIXED (p,q) where: p = 1 to 15 q = 0 to 15 | Yes |
| P | 16- 31 | 0 - 31 | CHARACTER (n) where n = length/2 + 1 | No |
| S | 1 - 15 | 0 - 15 | PICTURE '9...9V9...9R' | Yes |
| S | 16- 31 | 0 - 31 | CHARACTER (n) where n = 16 to 31 | No |
| F | 1 - 7 | 0 - 7 | DECIMAL FLOAT (7) UNALIGNED | Yes |

Figure 8-22 (Part 1 of 2). How DDS Data Types Are Defined in a PL/I Program

USING THE %INCLUDE DIRECTIVE

| DDS Data Type | Length | Decimal Position | Generated PL/I Declaration | Supported by PL/I |
|---------------|--------|------------------|--|-------------------|
| F | 8 - 15 | 0 - 15 | DECIMAL FLOAT (16) UNALIGNED where n = 1 to 15 | Yes |

Figure 8-22 (Part 2 of 2). How DDS Data Types Are Defined in a PL/I Program

DDS Features You Can Use in Your PL/I Program

ALIAS keyword

If you use the keyword `ALIAS` in your DDS, the name you specify as the `ALIAS` parameter is the name that the compiler generates in your program. The field name specified in the DDS is ignored. By using the `ALIAS`-name, you can make full use of PL/I's 31-character name length limit, with resulting improvements in program readability. You are not limited to a 10-character field-name. For an example of the use of the `ALIAS` keyword, see Figure 8-23 on page 8-79.

Indicators

The element type `INDICATORS` in the `%INCLUDE` directive generates a 99 byte structure for indicators. Each indicator defined in DDS generates a field declaration in the form `INnn`, where `nn` is the DDS indicator number. All other bytes in the structure, for which indicators have not been defined, generate field declarations in the form `INnn_INmm`, where `nn` is the first undefined byte, `mm` is the last undefined byte, and the bytes are contiguous. For example, if one indicator, 51, was defined, then the indicator structure would be:

```
IN01_IN50
IN51
IN52_IN99
```

Defined indicators will generate a `PICTURE '9'` and undefined indicators will generate `CHAR(n)`, where `n` represents the number of consecutive indicators not defined.

For indicators in the buffer, only the indicators that are used are declared. They are declared as `PICTURE '9'` and are generated when the element-type `INPUT`, `OUTPUT`, or `RECORD` is specified.

Key definitions

The element-type `KEY` in the `%INCLUDE` directive declares all the fields that are specified in the DDS as keys; their attributes are given in comments. For example, `/* ASCENDING */` is generated if the key field is ascending. Similarly, when you specify `INPUT`, `OUTPUT`, or `RECORD` in the `%INCLUDE` directive, a field-level comment indicates that the generated element is a key field in the DDS.

USING THE %INCLUDE DIRECTIVE

Prefixes

If you code a prefix-name in your %INCLUDE directive, the prefix you specify will be attached to the field name supplied by the file. For example, if you code

```
%INCLUDE STOCKFILE(COUNT,RECORD,CURRENT_)
```

and in the DDS the first field in the record is named

```
BANDSAWS
```

the generated name is

```
15 CURRENT_BANDSAWS
```

By using prefixes, you can generate meaningful names for different uses of the same record format inside a single program. For an example of the use of a prefix, see Figure 8-23.

TEXT keyword

The DDS keyword TEXT at the field level generates a comment line for the field declarations. Similarly, the keyword TEXT at the format level generates a comment line preceding the declarations taken from the DDS.

Sample Program Showing Use of DDS Features

| File | | Keying Instruction | Graphic Key | Description | Page of |
|------------|------|--------------------|-------------|-------------|---------|
| Programmer | Date | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 |
| 7 | 8 | 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 | 17 | 18 |
| 19 | 20 | 21 | 22 | 23 | 24 |
| 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | 32 | 33 | 34 | 35 | 36 |
| 37 | 38 | 39 | 40 | 41 | 42 |
| 43 | 44 | 45 | 46 | 47 | 48 |
| 49 | 50 | 51 | 52 | 53 | 54 |
| 55 | 56 | 57 | 58 | 59 | 60 |
| 61 | 62 | 63 | 64 | 65 | 66 |
| 67 | 68 | 69 | 70 | 71 | 72 |
| 73 | 74 | 75 | 76 | 77 | 78 |
| 79 | 80 | 81 | 82 | 83 | 84 |
| 85 | 86 | 87 | 88 | 89 | 90 |
| 91 | 92 | 93 | 94 | 95 | 96 |
| 97 | 98 | 99 | 100 | | |

| Sequence Number | Conditioning | | | | | Name | Length | Reference ID | Data Type (B/A/P/S/B/I/A/S/E/T/N/M/D/M) | Declared (Y/N) | Location | | Functions |
|-----------------|-------------------|------------------|--------------------|-------------------|---------------------------------------|-----------|--------|--------------|---|----------------|----------|-----|---|
| | Form Type (A/D/C) | Initials (A/D/C) | Particular (A/D/C) | Indicator (A/D/C) | Type of Name of Spec. (R/P/H/U/N/S/D) | | | | | | Line | Pos | |
| AA | | | | | R | CUSTFMT | | | | | | | |
| AA | | | | | | CUSTNAME | 25A | | | | | | TEXT('CUSTOMER NAME') |
| AA | | | | | | CUSTADDR1 | 25A | | | | | | ALIAS(CUSTOMER_HOME_ADDRESS) TEXT('CUSTOMER HOME ADDRESS') |
| AA | | | | | | CUSTADDR2 | 25A | | | | | | ALIAS(CUSTOMER_BUSINESS_ADDRESS) TEXT('CUSTOMER BUSINESS ADDRESS') |
| AA | | | | | | CUSTNO | 10S | | | | | | TEXT('CUSTOMER NUMBER') |
| AA | | | | | K | CUSTNO | | | | | | | UNSIGNED |
| A | | | | | | | | | | | | | |
| A | | | | | | | | | | | | | |
| A | | | | | | | | | | | | | |
| A | | | | | | | | | | | | | |
| A | | | | | | | | | | | | | |
| A | | | | | | | | | | | | | |
| A | | | | | | | | | | | | | |
| A | | | | | | | | | | | | | |

Figure 8-23 (Part 1 of 3). %INCLUDE Examples

USING THE %INCLUDE DIRECTIVE

```

5728PL1 R01 M00 880715          PL/I Source Listing          QTEMP/LP1430          11/30/88 15:31:28          Page 2
LP1430:
Include  SEQNBR STMT.SUBS BLK BN DO *<.....1.....2.....3.....4.....5.....6.....7.>.....8 Date
1      100    1          LP1430:
      200          PROC;
      300    2          1 1      DCL
      400          1 1
      500          1 A,
CUSTFMT + 100          /* %INCLUDE CUSTFILE(CUSTFMT,RECORD,PLANT_);
CUSTFMT + 200          /* ----- */
CUSTFMT + 300          /* PHYSICAL FILE: CUSTFILE.QTEMP
CUSTFMT + 400          /* FILE CREATION DATE: 87/11/30
CUSTFMT + 500          /* RECORD FORMAT: CUSTFMT
CUSTFMT + 600          /* RECORD FORMAT SEQUENCE ID: 371E00A681EA7
CUSTFMT + 700          /* ----- */
CUSTFMT + 800    2.1    1 1      6 15 PLANT_CUSTNAME CHAR(25), /* CUSTOMER NAME 7
CUSTFMT + 900    2.2    1 1      8 15 PLANT_CUSTOMER_HOME_ADDRESS CHAR(25),
CUSTFMT + 1000          /* CUSTOMER HOME ADDRESS 9
CUSTFMT + 1100    2.3    1 1      15 PLANT_CUSTOMER_BUSINESS_ADDRESS CHAR(25),
CUSTFMT + 1200          /* CUSTOMER BUSINESS ADDRESS
CUSTFMT + 1300    2.4    1 1      15 PLANT_CUSTNO PIC '999999999R';
CUSTFMT + 1400          /* CUSTOMER NUMBER
CUSTFMT 2      600    3          1 1      DCL
      700          1 B,
      800          /* %INCLUDE CUSTFILE(CUSTFMT,KEY);11
CUSTFMT + 100          /* ----- */
CUSTFMT + 200          /* PHYSICAL FILE: CUSTFILE.QTEMP
CUSTFMT + 300          /* FILE CREATION DATE: 87/11/30
CUSTFMT + 400          /* RECORD FORMAT: CUSTFMT
CUSTFMT + 500          /* RECORD FORMAT SEQUENCE ID: 371E00A681EA7
CUSTFMT + 600          /* ----- */
CUSTFMT + 700    3.1    1 1      15 CUSTNO PIC '999999999R';
CUSTFMT + 800          /* CUSTOMER NUMBER
CUSTFMT + 900          /* DDS - ASCENDING
CUSTFMT + 1000          /* 'UNSIGNED' KEY FIELD
CUSTFMT 900    4          1 1      DCL FILE1015 FILE RECORD UPDATE SEQL KEYED ENV(INTERACTIVE); LP100560
      1000    5          1 1      DCL 1 RFMT1, LP100590
      1100          /* %INCLUDE FILE1015 (SFLREC,RECORD);
SFLREC + 100          /* ----- */
SFLREC + 200          /* DEVICE FILE: FILE1015.QTEMP
SFLREC + 300          /* FILE CREATION DATE: 87/11/30
SFLREC + 400          /* RECORD FORMAT: SFLREC
SFLREC + 500          /* RECORD FORMAT SEQUENCE ID: 08C100874B716
SFLREC + 600          /* ----- */
SFLREC + 700          /* INDICATORS FOR FORMAT SFLREC
SFLREC + 800          /* INDICATOR 01 CHANGE INDICATOR
SFLREC + 900          /* INDICATOR 02 CHANGE INDICATOR
SFLREC + 1000          /* INDICATOR 03
SFLREC + 1100          /* ----- */
SFLREC + 1200    5.1    1 1      15 NUMBER1 PIC '99R',
SFLREC + 1300    5.2    1 1      15 ALPHA1 CHAR(10);
      1200    6          1 1      DCL 1 RFMT0,
      1300          /* %INCLUDE FILE1015 (SFLCTL,RECORD); 13
SFLCTL + 100          /* ----- */
SFLCTL + 200          /* DEVICE FILE: FILE1015.QTEMP
SFLCTL + 300          /* FILE CREATION DATE: 87/11/30

```

Figure 8-23 (Part 2 of 3). %INCLUDE Examples

USING THE %INCLUDE DIRECTIVE

```

5728PL1 R01 M00 880715          PL/I Source Listing          QTEMP/LP1430          11/30/88 15:31:28          Page 3
                                LP1430:
Include      SEQNBR STMT.SUBS BLK BN DO *<..+...1...+...2...+...3...+...4...+...5...+...6...+...7.>..+...8 Date
SFLCTL      + 400                      /* RECORD FORMAT: SFLCTL */
SFLCTL      + 500                      /* RECORD FORMAT SEQUENCE ID: 0230713C3E414 */
SFLCTL      + 600                      /* ----- */
SFLCTL      + 700                      /* INDICATORS FOR FORMAT SFLCTL */
SFLCTL      + 800                      /* INDICATOR 04 */
SFLCTL      + 900                      /* INDICATOR 05 */
SFLCTL      + 1000                     /* ----- */
SFLCTL      + 1100      6.1      1 1      14 15 DUMMYDCL CHAR(0); /* NO FIELDS OF NEEDED TYPE */
SFLCTL      1400      7      1 1      DCL 1 INDAREA, LP100730
SFLCTL      1500                      %INCLUDE FILE1015(SFLREC,INDICATORS); 15
SFLREC      + 100                      /* ----- */
SFLREC      + 200                      /* DEVICE FILE: FILE1015.QTEMP */
SFLREC      + 300                      /* FILE CREATION DATE: 87/11/30 */
SFLREC      + 400                      /* RECORD FORMAT: SFLREC */
SFLREC      + 500                      /* RECORD FORMAT SEQUENCE ID: 08C1008748716 */
SFLREC      + 600                      /* ----- */
SFLREC      + 700      7.1      1 1      16 15 IN01 PIC '9', /* CHANGE INDICATOR */
SFLREC      + 800      7.2      1 1      15 IN02 PIC '9', /* CHANGE INDICATOR */
SFLREC      + 900      7.3      1 1      15 IN03 PIC '9',
SFLREC      + 1000     7.4      1 1      16 15 IN04_IN99 CHAR(96); /* UNDEFINED INDICATOR(S) */
SFLREC      1600      8      1 1      DCL 1 INDAREA2, LP100730
SFLREC      1700                      %INCLUDE FILE1015(SFLCTL,INDICATORS);
SFLCTL      + 100                      /* ----- */
SFLCTL      + 200                      /* DEVICE FILE: FILE1015.QTEMP */
SFLCTL      + 300                      /* FILE CREATION DATE: 87/11/30 */
SFLCTL      + 400                      /* RECORD FORMAT: SFLCTL */
SFLCTL      + 500                      /* RECORD FORMAT SEQUENCE ID: 0230713C3E414 */
SFLCTL      + 600                      /* ----- */
SFLCTL      + 700      8.1      1 1      15 IN01_IN03 CHAR(03), /* UNDEFINED INDICATOR(S) */
SFLCTL      + 800      8.2      1 1      15 IN04 PIC '9',
SFLCTL      + 900      8.3      1 1      15 IN05 PIC '9',
SFLCTL      + 1000     8.4      1 1      15 IN06_IN99 CHAR(94); /* UNDEFINED INDICATOR(S) */
SFLCTL      1800      9      1 1      DCL INDICATORS(99) CHAR(1) BASED(PTR),
SFLCTL      1900      9.1      1 1      PTR POINTER;
SFLCTL      2000      10      1 1      DCL I BIN FIXED(15);
SFLCTL      2100      11      1 1      PTR = ADDR(INDAREA);
SFLCTL      2200      12      1 1      DO I=1 TO 99;
SFLCTL      2300      13      1 1 1      INDICATORS(I) = '0';
SFLCTL      2400      14      1 1 1      END;
SFLCTL      2500      15      1 1      PTR = ADDR(INDAREA2);
SFLCTL      2600      16      1 1      DO I=1 TO 99;
SFLCTL      2700      17      1 1 1      INDICATORS(I) = '0';
SFLCTL      2800      18      1 1 1      END;
SFLCTL      2900      19      1 1      OPEN FILE (FILE1015); LP100850
SFLCTL      3000      20      1 1      DO I=1 TO 5;
SFLCTL      3100      21      1 1 1      NUMBER1 = I;
SFLCTL      3200      22      1 1 1      ALPHA1 = 'XX';
SFLCTL      3300      23      1 1 1      WRITE FILE(FILE1015)FROM(RFMT1) KEYFROM(I)
SFLCTL      3400      1 1 1      OPTIONS(RECORD ('SFLREC'))
SFLCTL      3500      1 1 1      INDICATORS(INDAREA) );
SFLCTL      3600      24      1 1 1      IF IN03 = 1 THEN
SFLCTL      3700      1 1 1      IN03='0';
SFLCTL      3800      25      1 1 1      ELSE

```

Figure 8-23 (Part 3 of 3). %INCLUDE Examples

- 1** The source sequence field is set to 100 and is incremented by 100 with each line that follows.
- 2** The original line sequence numbers are resumed at the end of the generated statements.
- 3** The date field is left blank on generated statements. The actual %INCLUDE directive has a value put in the date field if it has been altered in SEU since the source was first entered.

USING THE %INCLUDE DIRECTIVE

- 4** The prefix `PLANT_` is specified as an option of the `%INCLUDE` directive.
- 5** Each occurrence of a `%INCLUDE` directive generates line comments that identify the file name, format name, record format sequence identifier, and file creation date.
- 6** The field name on the DDS is `CUSTNAME`. Because the prefix `PLANT_` was specified as an option of the `%INCLUDE` directive, the generated name is `PLANT_CUSTNAME`. The default level number for each generated field is 15.
- 7** The `TEXT` keyword was specified on the DDS with the parameter `CUSTOMER NAME`. The parameter is therefore printed as a comment after the declaration.
- 8** The field name on the DDS is `CUSTADDR1`. Because the keyword `ALIAS` was specified in the DDS with the parameter `CUSTOMER_HOME_ADDRESS`, the `ALIAS` parameter is used as the variable name and the prefix `PLANT_` is attached to it to generate the variable name `PLANT_CUSTOMER_HOME_ADDRESS`.
- 9** The keyword `TEXT` is also specified in the DDS with the parameter `CUSTOMER HOME ADDRESS`. The `TEXT` parameter is listed as a comment in the generated statement.
- 10** A comment is generated indicating that `CUSTNO` is defined on the DDS as a key.
- 11** `KEY` is specified as the element-type included from record format `CUSTFMT`.
- 12** A comment is generated indicating that `CUSTNO` is specified on the DDS as a key that is unsigned and in ascending sequence.
- 13** `RECORD` is specified as the element-type included from record format `SFLCTL`. Therefore, fields in the record that specify `BOTH` on the DDS are included in the generated statement. If `INDARA` is not specified on the DDS, indicators for which `BOTH` is specified on the DDS are also included in the generated statement.
- 14** Record format `SFLCTL` has no fields for which `BOTH` is specified on the DDS. The generated structure therefore contains no data elements. The compiler generates a character string `DUMMYDCL` with a length of zero.
- 15** `INDICATORS` is specified as the element-type included from record format `SFLREC`.
- 16** Indicators 1 to 3 are specified on the DDS, and are therefore declared in the generated text with `PIC '9'` as their attribute. Indicators 4-99 are not specified on the DDS, and are therefore not declared in the generated text. Instead, a variable `IN04_IN99` is declared in the generated text, and is given the attribute `CHARACTER` and a length of 96 bytes.

Chapter 9. References and Expressions

This chapter contains information on references and expressions, and their use in PL/I.

An **expression** is a representation of a value. It can consist of constants, variables, and function references, along with operators or parentheses, or both.

The syntax of expressions and references is shown in Figure 9-1 on page 9-2.

A reference can be a **scalar reference**, which refers to a scalar data item, an **array reference**, which refers to an array, or a **structure reference**, which refers to a structure. An expression that represents a scalar value is a **scalar expression**. The only non-scalar expressions are array references and structure references.

The syntax of many PL/I statements allows expressions. In this manual the term “expression” refers to a scalar expression except where stated to the contrary.

The examples that follow illustrate the syntax of expressions and references. They use the following declarations:

```
DECLARE BINFIXEDARRAY(10,10) BINARY FIXED (31),
        1 STRUCTURE1,
          5 DECFIXED1 DECIMAL FIXED (4,2),
          5 DECFIXED2 DECIMAL FIXED (4,2),
        1 STRUCTURE2(2),
          5 DECFIXED1 DECIMAL FIXED (4,2),
          5 DECFIXED2 DECIMAL FIXED (4,2),
        BINFIXED1 BINARY FIXED (15),
        BINFIXED2 BINARY FIXED (15),
        POINTER1 POINTER,
        ENTRYVAR ENTRY VARIABLE,
        ENTRYCON ENTRY;
```

Here are some examples of expressions:

BINFIXED1

A unary expression that is an elementary expression that is a reference that is a basic reference that is a name.

BINFIXED1BINFIXED2**

An expression that contains a unary expression followed by an infix operator followed by a second unary expression.

Because the expression contains an infix operator (the exponentiation operator ******), it is an operational expression. The unary expressions are operands. Operators are discussed under “Delimiters” on page 4-3. Operational expressions are discussed under “Operational Expressions” on page 9-4.

expression is:

unary-expression[*infix-operator*¹ *unary-expression*]...

where *unary-expression* is:

[*prefix-operator*²...] *elementary-expression*

where *elementary-expression* is:

(*expression*)|*constant*|*reference*

where *reference*³ is:

[*pointer-qualifier*] *basic-reference*
[(*subscript-list*)] [(*argument-list*)] [()]

where *pointer-qualifier*⁴ is:

reference – >

where *subscript-list*⁵ is:

expression [, *expression*] ...

where *argument-list*⁶ is:

[*expression* [, *expression*] ...]

where *basic-reference*⁷ is:

[*name*.] ... *name*

Notes:

¹Any of the operators shown in Figure 4-2 on page 4-4, except for the \rightarrow operator, can be used as an *infix-operator*; but \rightarrow , \leftarrow , \rightarrow , and \rightarrow are valid *infix-operators*.

²Any of the operators $+$, $-$, \rightarrow can be used as a *prefix-operator*.

³The optional empty argument list is discussed under “CALL Statement” on page 14-7.

⁴*pointer-qualifier* is discussed under “Based Variable Reference and Pointer Qualification” on page 5-20.

⁵*subscripts* are discussed under “Subscripts” on page 5-2.

⁶*arguments* are discussed in Chapter 14, “Procedures, Subroutines, and Functions.”

⁷Structure qualification (“.”) is discussed under “Structure-Qualification” on page 5-5.

Figure 9-1. Syntax of Expressions and References

-BINFIXED2

A unary expression that is a prefix operator followed immediately by a basic reference.

Because it contains the minus prefix operator, -, it is an operational expression; BINFIXED2 is the operand. The prefix operator specifies that the negative value of BINFIXED2 is used.

BINFIXEDARRAY

A basic reference.

Because BINFIXEDARRAY refers to an array, it is an array reference. Arrays are discussed under "Using Arrays and the Dimension Attribute" on page 5-1.

BINFIXEDARRAY(2,4)

A reference that is a basic reference followed by a subscript list.

BINFIXEDARRAY(2,4) is a subscripted reference.

2

A unary expression, an elementary expression, and a constant.

BINFIXEDARRAY(2,4) + BINFIXED1

An operational expression (see "Operational Expressions" on page 9-4).

(BINFIXEDARRAY(2,4) + BINFIXED1) * 2

An operational expression (see "Operational Expressions").

The sequence of operations is discussed under "Priority of Operators" on page 9-15.

POINTER1 - > BINFIXEDARRAY(2,4)

A pointer qualifier (POINTER1 - >), followed by a basic reference, followed by a subscript list.

It is a pointer-qualified and subscripted reference. Pointer-qualifiers are discussed under "Based Variable Reference and Pointer Qualification" on page 5-20.

STRUCTURE1.DECFIXED1

A basic reference which consists of a name, followed by a period, followed by a name.

STRUCTURE1.DECFIXED1 is a fully-qualified structure reference. Structures are discussed under "Using Structures and Level Numbers" on page 5-3.

STRUCTURE2.DECFIXED1(1)

A basic reference followed by a subscript list.

STRUCTURE2.DECFIXED1(1) is a fully-qualified and subscripted reference to a field of the first array element of STRUCTURE2. STRUCTURE2 is an array of structures. Arrays of structures are discussed under "Arrays of Structures" on page 5-5.

STRUCTURE1.DECFIXED1 - STRUCTURE2.DECFIXED1(1)

An operational expression (see "Operational Expressions" on page 9-4).

STRUCTURE1.DECFIXED1 / 4

An operational expression.

OPERATIONAL EXPRESSIONS

BINFIXEDARRAY(2,4) * STRUCTURE2.DECFIXED2(2)

An operational expression.

ENTRYVAR = ENTRYCON

An operational expression.

The remainder of this chapter discusses expressions that consist of operators and operands; such expressions are also called operational expressions.

Operational Expressions

An **operational expression** is an expression that consists of one or more operations. An **operation** can be either a **prefix operation**, which consists of a prefix operator followed by an operand, or an **infix operation**, which consists of an infix operator between two operands.

An operational expression can always be considered as a single operation. For example, the expression

$A + B * C$

is an infix operation whose operator is +, whose first operand is A, and whose second operand is $B * C$.

The operands and operator of an infix operation, or the operand and operator of a prefix operation, determine the attributes of the result of the operation. In this way, the attributes of the result of the entire operational expression can be determined.

The operands of an operation must be scalar expressions. The operation itself is also a scalar expression.

The operand(s) of an operation must be of the type required by the operator. For example, an arithmetic operator requires arithmetic operands. Therefore, before you use data as operands, you may have to convert them explicitly to the appropriate type, either by assigning them to a variable declared with the required attributes or, preferably, by means of the appropriate built-in function.

If the types of the operands are correct, some implicit conversion may still be necessary. For example, the operands of most arithmetic operators are converted to their common base and scale before the operation is processed. The types of the operands required by each operator and the conversions that will be processed implicitly are described below for each class of operations.

The rest of this chapter describes the four classes of operations: arithmetic, bit, comparison, and concatenation.

Arithmetic Operations

The operands of an arithmetic operation must be arithmetic.

You specify an arithmetic operation by one of the following operators:

+ - * / **

The + (plus) and - (minus) operators can be either prefix or infix operators, whereas the * (multiplication), / (division), and ** (exponentiation) operators are infix operators only.

Data Conversion in Arithmetic Operations

The two operands of an arithmetic operation may have different data attributes. They are converted as described below.

Picture data operands are considered fixed-point decimal operands with a precision (p,q), which is derived from the picture specification. They are treated as fixed-point decimal operands. The result of an arithmetic operation is always in coded arithmetic form.

Data conversion in arithmetic operations depends on if the operation is exponentiation:

- Operations other than exponentiation. The operands are converted, as necessary, to the base and scale of the result, which is the **common base and scale of the operands**, as follows:
 - If the bases of the two operands differ, the base of the result is binary.
 - If the scales of the two operands differ, the scale of the result is floating-point.
- Exponentiation. Three cases govern the data attributes of the result. The base, scale, and precision of the result depend on the significand and on the exponent, as shown in Figure 9-3 on page 9-8.

Results of Arithmetic Operations

After any necessary conversion of the operands, the arithmetic operation is processed.

The result of an operation must be floating-point, fixed-point binary with a zero scale factor, or fixed-point decimal with a non-negative scale factor. Therefore:

- If one operand of the divide operator is fixed-point binary, the other must not be fixed-point.
- If one operand of any infix operator, except exponentiation (**), or of the MAX, MIN, MOD, or DIVIDE built-in functions is fixed-point binary, the other operand must not be fixed-point decimal or picture with a nonzero scale factor.

OPERATIONAL EXPRESSIONS

The precision of the result is determined from its base and scale, from the precision of the operands after any necessary conversion, and from the operation.

Figure 9-2 on page 9-7 describes the result of the two steps involved in evaluating an operation: converting the operands to the base and scale of the result, and calculating the precision of the result. Figure 9-3 on page 9-8 shows the attributes of the result for the special cases of exponentiation noted in the right-hand column of Figure 9-2.

For the following example, refer to Figure 9-2. If the first operand is fixed-point binary with precision (5) and the second operand is fixed-point-decimal with precision (4), the result of multiplication is fixed-point binary with precision (21), which is calculated from the following:

$$p = 1 + p_1 + 1 + \text{ceil}(p_2 * 3.32)$$

where:

- p is the number-of-digits of the result
- p_1 is the number-of-digits of the first operand
- p_2 is the number-of-digits of the second operand.

Now refer to Figure 9-2 on page 9-7 and Figure 9-3 on page 9-8. If the same operands are used in an exponentiation, the result is floating-point binary with precision (5) from the first operand. In this example, case C applies.

With the DIVIDE built-in function, you can override the implementation precision rules for division.

In Figure 9-2, p_1 and q_1 , and p_2 and q_2 are the converted precisions of the operands, which have the base and scale of the result. ("ceil" means round the argument up to the next integer.) Figure 9-4 on page 9-8 is a table of $\text{ceil}(n*3.32)$ and $\text{ceil}(n/3.32)$ values.¹

The result of fixed-point division has the maximum implementation-defined number of digits, often with a large scale factor that may deny sufficient place for the integer part and therefore result in the overflow or truncation of the value. For example, the expression:

$$25+1/3$$

is evaluated as follows:

1. The division operation $1/3$ is processed first. (The sequence of operations is discussed under "Priority of Operators" on page 9-15.) This gives the intermediate result 0.333333333333333, which has the precision (15,14). Constants have the precision with which they are written.

¹ The number 3.32 is the approximate number of binary digits per decimal digit.

2. The integer 25, which has the precision (2,0), is added to the intermediate result, resulting in a precision of (15,14). The fixedoverflow condition is raised, and the result is undefined.

The following expression avoids truncation:

$$25+01/3$$

This expression is evaluated as follows:

1. The division operation is processed, giving the intermediate result 00.333333333333, which has the precision (15,13).
2. The integer 25 and the intermediate result are added, giving 25.333333333333.

You could alternatively use the DIVIDE built-in function:

$$25+DIVIDE(1,3,15,13)$$

See "DIVIDE(x,y,p[,q])" on page 15-10.

| First Operand | Second Operand | Attributes of the Result for Addition, Subtraction, Multiplication, or Division | Attributes of the Intermediate Result for Exponentiation | Addition or Subtraction Precision ³ | Multiplication Precision ² | Division Precision ³ |
|--|--|---|---|--|--|---|
| FIXED DECIMAL (p ₁ , q ₁) or PICTURE ¹ | FIXED DECIMAL (p ₂ , q ₂) | FIXED DECIMAL (p, q) | FLOAT DECIMAL (p) unless special case A or C applies ² p = max (p ₁ , q ₁) | p = 1 + max (p ₁ - q ₁ , p ₂ - q ₂) + q q = max (q ₁ , q ₂) | p = min(15, p ₁ + p ₂ + 1) q = q ₁ + q ₂ [q > b] | p = 15 q = 15 - ((p ₁ - q ₁) + q ₂) |
| | FIXED BINARY (p ₂) ^{4 5} | FIXED BINARY (p) | FLOAT BINARY (p) unless special case C applies ² p = max (ceil (p ₁ * 3.32), p ₂) | p = 1 + max (ceil (p ₁ * 3.32) + 1, p ₂) | p = ceil (p ₁ * 3.32) + p ₂ + 2 | |
| | FLOAT DECIMAL (p ₂) | FLOAT DECIMAL (p) | FLOAT DECIMAL (p) p = max (p ₁ , p ₂) | | p = max (p ₁ , p ₂) | |
| | FLOAT BINARY (p ₂) | FLOAT BINARY (p) | FLOAT BINARY (p) p = max (ceil (p ₁ * 3.32), p ₂) | | p = max (ceil (p ₁ * 3.32), p ₂) | |
| FIXED BINARY (p ₁) | FIXED DECIMAL (p ₂ , q ₂) or PICTURE ¹ | FIXED BINARY (p) | FLOAT BINARY (p) unless special case A or C applies ² p = max (ceil (p ₁ * 3.32), p ₂) | p = max (p ₁ , ceil (p ₂ * 3.32) + 1) + 1 | p = p ₁ + ceil (p ₂ * 3.32) + 2 | |
| | FIXED BINARY (p ₂) ⁵ | FIXED BINARY (p) | FLOAT BINARY (p) unless special case C applies ² p = max (p ₁ , p ₂) | p = 1 + max (p ₁ , p ₂) | p = p ₁ + p ₂ + 1 | |
| | FLOAT DECIMAL (p ₂) | FLOAT BINARY (p) | FLOAT BINARY (p) p = max (p ₁ , ceil (p ₂ * 3.32)) | | p = max (p ₁ , ceil (p ₂ * 3.32)) | |
| | FLOAT BINARY (p ₂) | FLOAT BINARY (p) | FLOAT BINARY (p) p = max (p ₁ , p ₂) | | p = max (p ₁ , p ₂) | |
| FLOAT DECIMAL (p ₁) | FIXED DECIMAL (p ₂ , q ₂) or PICTURE ¹ | FLOAT DECIMAL (p) | FLOAT DECIMAL (p) unless special case C applies ² p = max (p ₁ , p ₂) | | p = max (p ₁ , p ₂) | |
| | FIXED BINARY (p ₂) | FLOAT BINARY (p) | FLOAT BINARY (p) unless special case C applies ² p = max (ceil (p ₁ * 3.32), p ₂) | | p = max (ceil (p ₁ * 3.32), p ₂) | |
| | FLOAT DECIMAL (p ₂) | FLOAT DECIMAL (p) | FLOAT DECIMAL (p) p = max (p ₁ , p ₂) | | p = max (p ₁ , p ₂) | |
| | FLOAT BINARY (p ₂) | FLOAT BINARY (p) | FLOAT BINARY (p) p = max (ceil (p ₁ * 3.32), p ₂) | | p = max (ceil (p ₁ * 3.32), p ₂) | |
| FLOAT BINARY (p ₁) | FIXED DECIMAL (p ₂ , q ₂) or PICTURE ¹ | FLOAT BINARY (p) | FLOAT BINARY (p) unless special case C applies ² p = max (p ₁ , ceil (p ₂ * 3.32)) | | p = max (p ₁ , ceil (p ₂ * 3.32)) | |
| | FIXED BINARY (p ₂) | FLOAT BINARY (p) | FLOAT BINARY (p) unless special case C applies ² p = max (p ₁ , p ₂) | | p = max (p ₁ , p ₂) | |
| | FLOAT DECIMAL (p ₂) | FLOAT BINARY (p) | FLOAT BINARY (p) p = max (p ₁ , ceil (p ₂ * 3.32)) | | p = max (p ₁ , ceil (p ₂ * 3.32)) | |
| | FLOAT BINARY (p ₂) | FLOAT BINARY (p) | FLOAT BINARY (p) p = max (p ₁ , p ₂) | | p = max (p ₁ , p ₂) | |

Figure 9-2. Results of Arithmetic Operations

OPERATIONAL EXPRESSIONS

¹Picture data is fixed-point decimal. Its precision is derived from the PICTURE specification.

²Special cases of exponentiation are described in Figure 9-3.

³The calculations of precisions must not exceed the implementation maximums:

FIXED DECIMAL = 15
 FIXED BINARY = 31
 FLOAT DECIMAL = 16
 FLOAT BINARY = 53

⁴Except in exponentiation, the scale factor must be zero.

⁵The result of division is an integer: fractional digits are lost.

⁶The result of division must have a non-negative scale-factor.

ceil(x) means the smallest integer greater than or equal to x.

max(x,y) means the greater of x and y.

min(x,y) means the smaller of x and y.

| Case ¹ | First Operand | Second Operand | Attributes of Result |
|--|---|---|---|
| A | FIXED DECIMAL (p ₁ , q ₁) | Integer Constant with value n>0 | FIXED DECIMAL (p, q) [provided p≤15] p=(p ₁ + 1) * exp - 1 q=q ₁ * n |
| B | FIXED BINARY | Integer Constant with value n>0 | FIXED BINARY (p) [provided p≤31] p=(p ₁ + 1) * exp - 1 |
| C | FLOAT (p ₁) FIXED DECIMAL (p ₁ , q ₁) if neither case A or case B | FIXED (p ₂ , 0) or PICTURE without fractional part | FLOAT (p) with base of first operand p=max (p ₁ , p ₂) |
| ¹ If first operand = 0 and second operand > 0, the result is 0; if first operand = 0 and second operand ≤ 0, the ERROR condition is raised; if first operand < 0 and second operand is not an integer, the ERROR condition is raised. | | | |

Figure 9-3. Special Cases for Exponentiation

| n | ceil(n*3.32) | n | ceil(n/3.32) |
|----|--------------|-------|--------------|
| 1 | 4 | 1-3 | 1 |
| 2 | 7 | 4-6 | 2 |
| 3 | 10 | 7-9 | 3 |
| 4 | 14 | 10-13 | 4 |
| 5 | 17 | 14-16 | 5 |
| 6 | 20 | 17-19 | 6 |
| 7 | 24 | 20-23 | 7 |
| 8 | 27 | 24-26 | 8 |
| 9 | 30 | 27-29 | 9 |
| 10 | 34 | 30-33 | 10 |
| 11 | 37 | 34-36 | 11 |
| 12 | 40 | 37-39 | 12 |
| 13 | 44 | 40-43 | 13 |
| 14 | 47 | 44-46 | 14 |
| 15 | 50 | 47-49 | 15 |
| 16 | 54 | 50-53 | 16 |

Figure 9-4. Table of Ceil Values

Bit Operations

The operands of a bit operation must be bit strings. If the operands of an infix operation differ in length, the shorter operand is padded on the right with zeros.

You specify a bit operation by one of the following operators:

¬ & |

You can use the ¬ operator (“not”) only as a prefix operator, the & (“and”) and | (“or”) operators only as infix operators.

The result of a bit operation is a bit value equal in length to the longer operand.

Bit operations are processed bit-by-bit, the operators having the same function as in boolean algebra. The results are as follows:

- ¬ The bits are reversed: '1'B becomes '0'B and '0'B becomes '1'B.
- & If both corresponding bits are '1'B, the result is '1'B; otherwise, the result is '0'B.
- | If both corresponding bits are '0'B, the result is '0'B; otherwise, the result is '1'B.

OPERATIONAL EXPRESSIONS

The following table shows the result for each bit position for each of the operators:

| A | B | \neg A | \neg B | A&B | A B |
|---|---|----------|----------|-----|-----|
| 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 |

The following examples use the following operand values:

| Operand | Value |
|---------|-----------|
| ABIT | '010111'B |
| BBIT | '111111'B |
| CBIT | '110'B |

The following operators produce the following results:

| Operation | Results |
|--|-----------|
| \neg ABIT | '101000'B |
| \neg CBIT | '001'B |
| CBIT & BBIT | '110000'B |
| ABIT BBIT | '111111'B |
| CBIT BBIT | '111111'B |
| ABIT (\neg CBIT) | '1'B. |
| \neg ((\neg CBIT) (\neg BBIT)) | '110111'B |

Due to the padding required for CBIT, the operations CBIT & BBIT and \neg ((\neg CBIT)|(\neg BBIT)) are not equivalent.

Comparison Operations

Both operands of a comparison operator must be arithmetic, or both must be character, or both must be bit, or they must both be of the same program control data type. Comparisons of entry, file, pointer, or label data can use only the = or \neg = operators.

You specify a comparison operation by one of the following operators:

| Operator | Meaning |
|----------|--------------------------|
| < | less than |
| \neg < | not less than |
| < = | less than or equal to |
| = | equal to |
| \neg = | not equal to |
| > = | greater than or equal to |
| > | greater than |
| \neg > | not greater than |

Comparisons of **problem data** can be:

- **Arithmetic**, which involves the comparison of signed numeric values. If the operands differ in base or scale, they are converted to their common base and scale, as described under “Data Conversion in Arithmetic Operations” on page 9-5. Picture data operands are considered fixed-point decimal.

If one operand is fixed-point binary, the other operand must not be fixed-point decimal or pictured with a nonzero scale-factor.

- **Character**, which involves left-to-right, character-by-character comparison according to the collating sequence. Where the lengths of the operands differ, the shorter is padded on the right with blanks.
- **Bit**, which involves left-to-right, bit-by-bit comparison of binary digits. The shorter is padded on the right with zeros.

Comparisons of **program control data** are equal when:

- **File** operands represent the same file constant.
- **Label** operands refer to the same statement in the same block activation.
- **Pointer** operands have the same value.
- **Entry** operands refer to the same entry name and, for internal entry constants, their values refer to the same block activation.

The result of a comparison operation is a bit value of length 1; the value is '1'B if the relationship is true, or '0'B if the relationship is false.

An example of a comparison operation in an IF statement is:

```
IF A = B
THEN action-if-true
ELSE action-if-false
```

The evaluation of the expression `A = B` yields either '1'B (true) or '0'B (false); the action-if-true or action-if-false will be taken accordingly.

In the assignment statement

```
X = A < B;
```

the value '1'B would be assigned to X if A were less than B; otherwise, the value '0'B would be assigned.

In the following example, the value of CBIT is '110111'B and FFID is 5:

```
SUBSTR(CBIT,1,1) | (FFID=5)
```

The result of `FFID = 5` is '1'B, as is the result of `SUBSTR(CBIT,1,1)`. The result of the “or” operation is therefore '1'B. If this expression appears in an IF statement, the length of the first operand must be 1.

Table for Comparison Operations

Figure 9-5 on page 9-13 shows the attributes to which the two operands of a comparison operation are converted before they are compared. These conversions are derived from the rules given on the preceding pages. (Ceil values are given in Figure 9-4 on page 9-9.)

Find the row in Figure 9-5 that corresponds to the two operands in the expression evaluated. The first column refers to the first operand and the second to the second operand. The third and fourth columns give the attributes of the intermediate targets of the first and second operands, respectively, after any necessary conversion.

For example:

```
DECLARE ITEM PICTURE '99999',
          STANDARD FIXED BINARY(15);
IF ITEM=STANDARD THEN DO;
```

In Figure 9-5, the entries in the third and fourth columns that correspond to a first operand with the PICTURE attribute and a second operand with the attributes FIXED BINARY are FIXED BINARY (1 + ceil($p_1 * 3.32$)). This indicates that ITEM is converted to coded arithmetic form with the first set of attributes and that STANDARD is not converted. The precision (p_1) is derived from the picture specification and (p_2) from FIXED BINARY (15) respectively.

The tables indicate that ITEM will be converted to FIXED BINARY (18) and then compared algebraically with STANDARD, whose attributes remain FIXED BINARY (15).

| First Operand | Second Operand | Attributes after Conversion | |
|--|---|---|--|
| | | First Operand | Second Operand |
| FIXED DECIMAL (p ₁ , q ₁) or PICTURE ¹ | FIXED DECIMAL (p ₂ , q ₂) ^{1,2} | FIXED DECIMAL (p ₁ , q ₁) | FIXED DECIMAL (p ₂ , q ₂) |
| | PICTURE ² | | |
| | FIXED BINARY (p ₂) | FIXED BINARY | FIXED BINARY (p ₂) |
| | FLOAT DECIMAL (p ₂) | FLOAT DECIMAL (p ₁) | FLOAT DECIMAL (p ₂) |
| | FLOAT BINARY (p ₂) | FLOAT BINARY (p ₁) (1+ceil (p ₁ *3.32)) | FLOAT BINARY (p ₂) |
| FIXED BINARY (p ₁) | FIXED DECIMAL (p ₂ , q ₂) | FIXED BINARY (p ₁) | FIXED BINARY (1+ceil (p ₂ *3.32)) |
| | PICTURE ² | | FIXED BINARY (p ₂) |
| | FIXED BINARY (p ₂) | | |
| | FLOAT DECIMAL (p ₂) | FLOAT BINARY | FLOAT BINARY (ceil (p ₂ *3.32)) |
| | FLOAT BINARY (p ₂) | | FLOAT BINARY (p ₂) |
| FLOAT DECIMAL (p ₁) | FIXED DECIMAL (p ₂ , q ₂) | FLOAT DECIMAL (p ₁) | FLOAT DECIMAL (p ₂) |
| | PICTURE ¹ | | |
| | FIXED BINARY (p ₂) | FLOAT BINARY | FLOAT BINARY (p ₂) |
| | FLOAT DECIMAL (p ₂) | FLOAT DECIMAL (p ₁) | FLOAT DECIMAL (p ₂) |
| | FLOAT BINARY (p ₂) | FLOAT BINARY ceil (p ₁ *3.32)) | FLOAT BINARY (p ₂) |
| FLOAT BINARY (p ₁) | FIXED DECIMAL (p ₂ , q ₂) | FLOAT BINARY (p ₁) | FLOAT BINARY (ceil (p ₂ *3.32)) |
| | PICTURE ² | | FLOAT BINARY (p ₂) |
| | FIXED BINARY (p ₂) | | FLOAT BINARY (ceil (p ₂ *3.32)) |
| | FLOAT DECIMAL (p ₂) | | |
| | FLOAT BINARY (p ₂) | | FLOAT BINARY (p ₂) |
| CHARACTER (n ₁) | CHARACTER (n ₂) | CHARACTER (max(n ₁ , n ₂)) | CHARACTER (max (n ₁ , n ₂)) |
| BIT (n ₁) | BIT (n ₂) | BIT (max (n ₁ , n ₂)) | BIT (max (n ₁ , n ₂)) |

¹ If you specify a scale factor, it must be equal to zero.

² Picture data is fixed point decimal. Its precision is derived from the PICTURE specification.

ceil (x) is the smallest integer that is greater than or equal to x.
max (x,y) is the greater of x and y.

Figure 9-5. Results of Comparison Operations

OPERATIONAL EXPRESSIONS

Concatenation Operations

Concatenation can be processed only upon strings. Both strings must be character or both must be bit.

You specify a concatenation operation by the concatenation operator

||

It signifies that the operands are to be joined in such a way that the last character or bit of the operand to the left immediately precedes the first character or bit of the operand to the right with no intervening bits or characters.

The result of concatenating two character strings is a character string, and the result of concatenating two bit strings is a bit string; the length of the resulting string is the sum of the lengths of the two operands.

For example, concatenation operations using the following operands and values:

| Operand | Value |
|---------|-----------|
| ABIT | '010111'B |
| BBIT | '101'B |
| CCHAR | 'XY,Z' |
| DCHAR | 'AA/BB' |

produce the following results:

| Operation | Results |
|------------------|--------------------|
| ABIT BBIT | '010111101'B |
| ABIT ABIT BBIT | '010111010111101'B |
| CCHAR DCHAR | 'XY,ZAA/BB' |
| DCHAR CCHAR | 'AA/BBXY,Z' |

Combinations of Operations

You can combine different operations within the same expression, provided you observe the rules for the data types of the operands for each operator. The result of each embedded operation is used as an operand of a further operation. For example:

```
DECLARE RESULT BIT (3) ALIGNED,  
             AFID  FIXED DECIMAL (1) STATIC INITIAL (2),  
             BFIB  FIXED BINARY (3) STATIC INITIAL (6),  
             CFLOD FLOAT DECIMAL (2) STATIC INITIAL (32000),  
             DBIT  BIT (4) ALIGNED STATIC INITIAL ('1101'B);  
AFID = 2;  
BFIB = 6;  
CFLOD = 32000;  
DBIT = '1101'B;  
RESULT = AFID + BFIB < CFLOD & DBIT;
```

The operands are converted, as required, before each operation is processed in the following order:

1. The decimal value of AFID is converted to a binary value of 2.
2. The value of BFIB is added to the converted value of AFID and stored in a halfword binary fixed intermediate result, which therefore has a value of 8.
3. CFLOD and the fixed-point binary result of AFID + BFIB are converted to floating-point binary, and the intermediate results are compared.
4. The result of the comparison, a bit value of length 1 with a value of '1' is extended with zeros to the length of the bit variable DBIT, and the "and" operation is processed.
5. The result of the "and" operation is a bit value of length 4 with a value of '1000'B. It is assigned to RESULT without conversion, but with truncation on the right; the final value is therefore '100'B.

Although in this example the expression is evaluated operation-by-operation, from left to right, the order of evaluation is in fact determined by the priority of the operators.

Priority of Operators

The order in which operations in an expression are processed is based on the priority of the operators involved: the operation with the operator of the highest priority is processed first, that with the operator of the lowest priority last. The priority of the operators in the evaluation of expressions is shown in Figure 9-6.

| Priority | Operator | Type of Operation |
|----------|-------------------------|-------------------|
| 1 | ** | arithmetic |
| | prefix + prefix - | arithmetic |
| | ¬ | bit |
| 2 | * / | arithmetic |
| 3 | infix + infix - | arithmetic |
| 4 | | concatenation |
| 5 | < ¬< <= = ¬= >= > ¬> | comparison |
| 6 | & | bit |
| 7 | | bit |

Figure 9-6. Priority of Operators

The operators are listed in order of priority, 1 being the highest priority and 7 the lowest. Operators in the same group have the same priority. For example, the exponentiation operator ** has the same priority as the prefix +, prefix -, and ¬ operators.

OPERATIONAL EXPRESSIONS

If two or more operators of priority 1 appear in an expression, their order of evaluation is from right to left; that is, the rightmost exponentiation or prefix operator is applied first, the next rightmost next, and so on.

For all other operators, if two or more operators of the same priority appear in an expression, their order of evaluation is from left to right.

For example:

```
DECLARE RESULT FIXED DECIMAL (7,1),
          AFID  FIXED DECIMAL (1),
          BFIB  FIXED BINARY (3),
          CFLOD FLOAT DECIMAL (2),
          DPIC  PICTURE '999V9';
RESULT = AFID * BFIB * CFLOD + DPIC;
```

The operations in this expression are processed from left to right, because multiplication has a higher priority than addition, and the order of evaluation of the two multiplications is from left to right.

The order of evaluation (and, consequently, the result) of an expression can be changed by parentheses. Expressions enclosed in parentheses are evaluated first, starting with the innermost parenthesized expression, before they are considered in relation to surrounding operators.

The expression above is evaluated as if its operations were parenthesized as follows:

```
((AFID * BFIB) * CFLOD) + DPIC
```

The following expression

```
AFID * (BFIB * (CFLOD + DPIC))
```

is evaluated as follows. The value of DPIC would be converted to floating-point decimal and added to the value of CFLOD, yielding a floating-point decimal result (result_1). The value of BFIB and result_1 would then be converted to floating-point binary and multiplied, yielding a floating-point binary result (result_2). The value of AFID would then be converted to floating-point binary and multiplied by result_2, yielding a floating-point binary result.

For parenthesized expressions within expressions, PL/I specifies only that the parenthesized expressions will be evaluated before any unparenthesized expression. It does not specify the order in which the parenthesized expressions will be evaluated.

For example:

```
(AFID + BFIB) * (CFLOD - DPIC)
```

In this case, the parenthesized addition and subtraction will be done before the unparenthesized multiplication. PL/I does not specify if (AFID + BFIB) or (CFLOD - DPIC) will be evaluated first.

Chapter 10. Condition Handling Statements

While a program is running, certain exceptional situations, called **conditions**, may be detected. When a condition is detected, it is said to be **raised**. These conditions may be errors, such as overflow, or they may be expected situations, such as the end of an input file.

When a condition is raised, the action established for it is run. For each condition listed under “Specifiable Conditions in ON and SIGNAL Statements,” you can either use the ON statement to establish the action taken or rely on the implicit action defined for it.

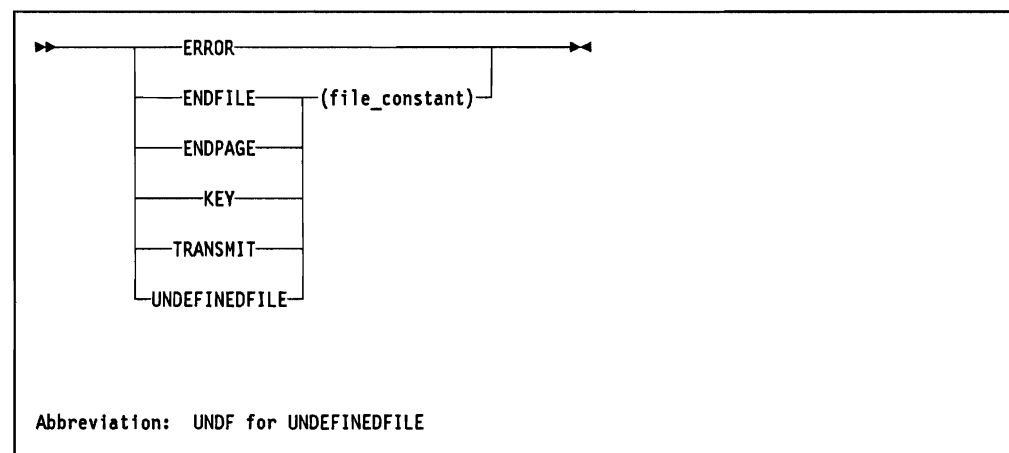
For each condition, there are condition codes that correspond to the different situations in which the condition can be raised. You can obtain the condition code and other information associated with a raised condition by using the condition-handling built-in functions in an on-unit.

In certain cases, conditions may be detected while compiling your program. These are diagnosed and are not raised when the program is run.

The following sections summarize the conditions for which ON and SIGNAL statements can be written, as well as other conditions that may be raised. Each condition is discussed, together with its implicit action and condition codes, in Appendix D, “Conditions and Condition Codes.”

Specifiable Conditions in ON and SIGNAL Statements

Two groups of conditions can be specified in an ON or SIGNAL statement: the ERROR condition and the input/output conditions.



The ERROR condition is raised when any one of several errors is detected. It is also raised as the implicit action for a number of other conditions, such as the conversion condition. You can find out which condition was raised, by using the ONCODE built-in function in an on-unit. (On-units are discussed under “ON Statement” on page 10-2.)

ON STATEMENT

Input/output conditions always relate to a particular file. For example, there is an ENDFILE condition for each file used in the program.

Unspecifiable Conditions

Conditions that will be detected, but for which you cannot write ON or SIGNAL statements, are:

- conversion
- fixedoverflow
- overflow
- record(file-constant)
- storage
- stringize
- underflow
- zerodivide

When one of these conditions is detected, the implicit action (see below) is taken.

Established Action

The **established action** for a condition is the action taken if the condition is raised. This is either the implicit action or the action specified in an ON statement for the condition.

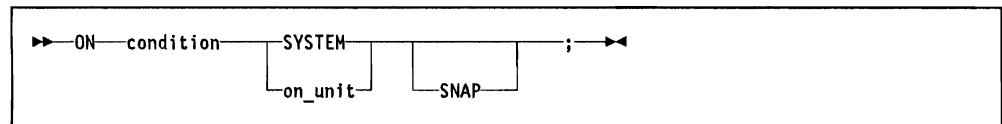
Implicit Action

Each condition has an **implicit action**, which is established when the program is activated and remains established unless overridden by processing an ON statement for the same condition (see "Scope of the Established Action" on page 10-4).

The implicit action for most conditions is to issue a message and raise the ERROR condition. The implicit action for each condition is given in Appendix D, "Conditions and Condition Codes."

ON Statement

The ON statement explicitly establishes the action taken when a condition is raised. The action established by the ON statement overrides or suspends any currently established action unless overridden by a further ON statement for the same condition, or until the block it was processed in ends (see "Scope of the Established Action" on page 10-4). The ON statement can also be used to reestablish the implicit action.



condition

Any of the conditions given under "Specifiable Conditions in ON and SIGNAL Statements" on page 10-1.

 IBM Extension

SNAP

Transmits diagnostic data to the AS/400 dump file QPPGMDMP. The data includes:

- The current statement number.
- A list of currently active blocks and on-units, listed in order of calling.

SYSTEM

Specifies that the implicit action is established for the condition.

 End of IBM Extension

on_unit

Specifies the action established for the condition. It can be either an unlabeled statement (including a null statement), or an unlabeled begin-block. If it is an unlabeled statement, the following statements are not allowed: BEGIN, DECLARE, DO, END, IF, ITERATE, LEAVE, PROCEDURE, RETURN, or SELECT. If it is a begin-block, a RETURN statement can appear only in a procedure nested within the block.

Because the on-unit itself requires a semicolon, no semicolon is shown for the on-unit in the syntax.

Up to 49 on-units can be concurrently active in any block. An external procedure can contain up to 254 on-units.

An on-unit is treated as a procedure without parameters that is internal to the block in which it appears. Therefore, any names referenced in an on-unit are those known in the block in which the ON statement was processed, rather than in the block in which the condition was raised.

Running an On-Unit

An on-unit runs when the specified condition is raised, not when the ON statement is processed.

An on-unit ends normally by returning control to the block from which the on-unit was entered. Control will return to the statement immediately after the statement that raised the condition if the on-unit ends normally.

Running an on-unit may be ended abnormally by a GO TO statement that transfers control out of the on-unit, which allows program processing to continue, or by a STOP statement, which ends the run unit.

The point to which control normally returns depends on the condition. Control may return to the point immediately following the point at which the condition was raised or to the statement following the one in which the condition was raised.

The effect of a null statement on-unit is to process normal return from the on-unit.

If an ERROR on-unit ends normally, the program will end abnormally.

SIGNAL STATEMENT

For more information about on-units and condition handling, see Appendix D, “Conditions and Condition Codes.”

Scope of the Established Action

The established action remains in effect throughout the block in which the ON statement was processed and throughout all dynamically descendant blocks, unless it is overridden by an ON statement for the same condition or until the block in which the ON statement was processed ends.

When another ON statement for the same condition is processed, the established action is affected as follows:

- If the block that contains the new ON statement is a dynamic descendant of the block that contains the earlier ON statement, the action established by the earlier ON statement is suspended.

When control returns to the block that contains the earlier ON statement, all actions that were current immediately before its suspension are reestablished. Therefore, no subroutine can change the established action for its calling block.

- If both ON statements are processed in the same block activation, the earlier ON statement is overridden. The earlier action can be reestablished only by processing an appropriate ON statement.

Scope of Values of Condition Handling Built-In Functions

The value of a condition handling built-in function is set when an on-unit for the corresponding condition is entered. This is described separately for each of the condition handling built-in functions (see Chapter 15, “Built-In Functions, Subroutines, and Pseudovariables”). The value remains in effect throughout the processing of that on-unit and its dynamic descendants, unless it is temporarily overridden when a second on-unit for a condition specific to the function is entered. If the second on-unit ends, the values that were in effect before it was entered are reinstated.

SIGNAL Statement

The SIGNAL statement raises a specified condition. It simulates the occurrence of the condition and forces the processing of the established action.

```
▶▶—SIGNAL—condition;—▶▶
```

condition

Any of the conditions given under “Specifiable Conditions in ON and SIGNAL Statements” on page 10-1.

Example of Use of Conditions

The subroutine shown in the example below illustrates the use of the ON statement and the SIGNAL statement. It reads records from a file SEQFILE. Each record consists of two values that are used to process the second file INFILE.

| 5728PL1 R01 M00 880715 | PL/I Source Listing | PLITST/LP1431 | 11/30/87 14:51:09 | Page 2 |
|------------------------|---|----------------------------|-------------------|-----------------|
| Include | LP1431: PROCEDURE; | | | PUB00160 |
| | LP1431: PROCEDURE; | | | PUB00160 850109 |
| 100 | | | | PUB00170 |
| 200 | | | | PUB00180 |
| 300 | /* FILE DECLARATIONS */ | | | PUB00190 |
| 400 | DECLARE | | | PUB00200 841113 |
| 500 | INFILE FILE RECORD INTERNAL SEQUENTIAL KEYED INPUT | | | 841113 |
| 600 | ENV(CONSECUTIVE), | | | 841113 |
| 700 | SEQFILE FILE RECORD INTERNAL SEQUENTIAL INPUT ENV(CONSECUTIVE), | | | 841114 |
| 800 | SYSPRINT FILE STREAM OUTPUT PRINT, | | | 841114 |
| 900 | CALC_PROC EXTERNAL ENTRY; | | | 841114 |
| 1000 | | | | PUB00220 |
| 1100 | /* RECORD DECLARATIONS */ | | | PUB00230 |
| 1200 | DECLARE | | | PUB00240 |
| 1300 | 1 RELATIVE_RECORD, | | | PUB00260 841107 |
| 1400 | 2 REL_WEEK | PICTURE '99', | | PUB00280 841107 |
| 1500 | 2 REL_UNIT_SALES | PICTURE 'S999999'; | | PUB00290 841113 |
| 1600 | | | | 841107 |
| 1700 | DECLARE | | | 841107 |
| 1800 | 1 INPUT_RECORD, | | | 841107 |
| 1900 | 2 START_WEEK | BINARY FIXED(15), | | 841107 |
| 2000 | 2 NUMB_WEEKS | BINARY FIXED(15); | | 841107 |
| 2100 | | | | PUB00340 |
| 2200 | /* PROGRAM FLAGS */ | | | PUB00350 |
| 2300 | DECLARE | | | PUB00360 |
| 2400 | 1 BIT_FLAGS STATIC, | | | PUB00370 |
| 2500 | 2 MORE_RECORDS | BIT(1) ALIGNED, | | PUB00380 |
| 2600 | 2 ERR_KEY | BIT(1) ALIGNED, | | 841113 |
| 2700 | 2 NO | BIT(1) ALIGNED INIT('0'B), | | PUB00390 |
| 2800 | 2 YES | BIT(1) ALIGNED INIT('1'B); | | PUB00400 |
| 2900 | | | | PUB00410 |
| 3000 | /* PROGRAM VARIABLES */ | | | PUB00420 |
| 3100 | DECLARE | | | PUB00430 |
| 3200 | KEY_DATA | BINARY FIXED(15), | | PUB00440 841108 |
| 3300 | ONCODE | BUILTIN; | | 841108 |
| 3400 | | | | 841113 |
| 3500 | | | | PUB00460 |
| 3600 | /* MAIN PROGRAM */ | | | PUB00560 841113 |
| 3700 | 7 1 1 1 ON ENDFILE(SEQFILE) | | | PUB00470 841113 |
| 3800 | 1 1 MORE_RECORDS = NO; | | | PUB00480 |
| 3900 | | | | PUB00550 |
| 4000 | /* ESTABLISH ACTION FOR ERROR IN DIST */ | | | 841113 |
| 4100 | 8 1 1 2 ON ERROR BEGIN; | | | 841107 |
| 4200 | 9 3 2 ON ERROR SYSTEM; | | | 841107 |
| 4300 | 10 3 2 IF ERR_KEY = YES THEN | | | 841113 |
| 4400 | 3 2 DO; | | | 841113 |
| 4500 | 11 3 2 1 PUT FILE(SYSPRINT) SKIP EDIT('INVALID START WEEK:',START_WEEK) | | | 841113 |
| 4600 | 3 2 1 (A,F(3)); | | | 841113 |
| 4700 | 12 3 2 1 GOTO END_PROCI; | | | 841113 |
| 4800 | 13 3 2 1 END; | | | 841113 |
| 4900 | 14 3 2 ELSE DO; | | | 841113 |
| 5000 | 15 3 2 1 PUT FILE(SYSPRINT) SKIP | | | 841113 |
| 5100 | 3 2 1 EDIT('ERROR IN PROCESSING PROCEDURE: DIST')(A); | | | 841113 |
| 5200 | 16 3 2 1 GOTO END_PROCI; | | | 841113 |
| 5300 | 17 3 2 1 END; | | | 841113 |

Figure 10-1 (Part 1 of 2). Use of the ON and SIGNAL Statements

SIGNAL STATEMENT

| 5728PL1 R01 M00 880715 | PL/I Source Listing | PLITST/LP1431 | 11/30/88 14:51:09 | Page 3 | | | |
|------------------------|---------------------|---------------|-------------------|----------|----|---|-----------------|
| Include | LP1431: PROCEDURE; | | | PUB00160 | | | |
| SEQNBR | STMT. | SUBS | BLK | BN | DO | *<...1...2...3...4...5...6...7...>...8 | Date |
| 5400 | 18 | | 3 | 2 | | END; /* BEGIN */ | 841113 |
| 5500 | | | | | | | 841107 |
| 5600 | | | | | | | 841107 |
| 5700 | 19 | | 1 | 1 | 1 | MORE_RECORDS = YES; | PUB00570 |
| 5800 | 20 | | 1 | 1 | | ERR_KEY = NO; | 841113 |
| 5900 | | | | | | | PUB00580 |
| 6000 | 21 | | 1 | 1 | | OPEN | PUB00590 841107 |
| 6100 | | | 1 | 1 | | FILE (SEQFILE); /* INPUT */ | PUB00600 841113 |
| 6200 | 22 | | 1 | 1 | | OPEN | 841107 |
| 6300 | | | 1 | 1 | | FILE (INFILE); /* INPUT */ | PUB00600 841113 |
| 6400 | | | | | | | 841113 |
| 6500 | 23 | | 1 | 1 | | READ FILE (SEQFILE) INTO (INPUT_RECORD); | PUB00710 841113 |
| 6600 | | | | | | | PUB00650 |
| 6700 | 24 | | 1 | 1 | 1 | DO WHILE(MORE_RECORDS); | 841107 |
| 6800 | 25 | | 1 | 1 | 1 | IF START_WEEK <1 START_WEEK >52 THEN | 841113 |
| 6900 | | | 1 | 1 | 1 | DO; | 841113 |
| 7000 | 26 | | 1 | 1 | 2 | ERR_KEY = YES; | 841113 |
| 7100 | 27 | | 1 | 1 | 2 | SIGNAL ERROR; | 841113 |
| 7200 | 28 | | 1 | 1 | 2 | END; | 841113 |
| 7300 | 29 | | 1 | 1 | 1 | CALL RANDOM; | PUB00680 841107 |
| 7400 | 30 | | 1 | 1 | 1 | READ FILE (SEQFILE) INTO (INPUT_RECORD); | PUB00710 841113 |
| 7500 | 31 | | 1 | 1 | 1 | END; /* DO WHILE */ | PUB00720 |
| 7600 | | | | | | | PUB00730 |
| 7700 | | | | | | | PUB00780 |
| 7800 | 32 | | 1 | 1 | | RANDOM: PROCEDURE; | PUB00790 |
| 7900 | | | | | | | 841107 |
| 8000 | | | | | | /* ESTABLISH ACTION FOR ERROR IN RANDOM */ | 841113 |
| 8100 | 33 | | 4 | 2 | 1 | ON ERROR BEGIN; | 841107 |
| 8200 | 34 | | 5 | 3 | | ON ERROR SYSTEM; | 841107 |
| 8300 | 35 | | 5 | 3 | | PUT FILE(SYSPRINT) SKIP | 841107 |
| 8400 | | | 5 | 3 | | EDIT('ERROR IN PROCESSING PROCEDURE: RANDOM')(A); | 841107 |
| 8500 | 36 | | 5 | 3 | | GOTO END_PROCL; | 841107 |
| 8600 | 37 | | 5 | 3 | | END; /* BEGIN */ | 841107 |
| 8700 | | | | | | | 841107 |
| 8800 | | | | | | /* ESTABLISH ACTION FOR KEY IN RANDOM */ | 841113 |
| 8900 | 38 | | 4 | 2 | 1 | ON KEY(INFILE) BEGIN; | PUB00500 841113 |
| 9000 | 39 | | 6 | 3 | | ON ERROR SYSTEM; | PUB00510 841107 |
| 9100 | 40 | | 6 | 3 | 1 | IF ONCODE = 51 THEN | PUB00520 841107 |
| 9200 | | | 6 | 3 | | DO; | PUB00530 841107 |
| 9300 | 41 | | 6 | 3 | 1 | PUT FILE(SYSPRINT) SKIP | 841107 |
| 9400 | | | 6 | 3 | 1 | EDIT('SPECIFIED KEY NOT FOUND IN FILE INFILE, KEY=',KEY_DATA | 841113 |
| 9500 | | | 6 | 3 | 1 | (A,A); | 841113 |
| 9600 | 42 | | 6 | 3 | 1 | END; | 841107 |
| 9700 | 43 | | 6 | 3 | 1 | ELSE | 841107 |
| 9800 | | | 6 | 3 | | PUT FILE(SYSPRINT) SKIP | 841107 |
| 9900 | | | 6 | 3 | | EDIT('ERROR ON KEY SPECIFIED, KEY=',KEY_DATA)(A,A); | 841107 |
| 10000 | 44 | | 6 | 3 | | END; /* BEGIN */ | PUB00540 841107 |
| 10100 | | | | | | | 841107 |
| 10200 | 45 | | 4 | 2 | 1 | DO KEY_DATA = START_WEEK TO (START_WEEK + NUMB_WEEKS-1) BY 1; | 841113 |
| 10300 | 46 | | 4 | 2 | 1 | READ FILE (INFILE) INTO (RELATIVE_RECORD) KEY (KEY_DATA); | PUB00800 841113 |
| 10400 | 47 | | 4 | 2 | 1 | CALL CALC_PROC; | 841113 |
| 10500 | 48 | | 4 | 2 | 1 | END; /* DO LOOP */ | PUB00840 841107 |
| 10600 | | | | | | | 841107 |
| 10700 | 49 | | 4 | 2 | | END RANDOM; | PUB00850 |
| 10800 | | | | | | | PUB00860 |
| 10900 | 50 | | 1 | 1 | | END_PROCL: | 841107 |
| 11000 | | | 1 | 1 | | CLOSE | PUB00740 841107 |
| 11100 | | | 1 | 1 | | FILE (SEQFILE); | PUB00750 841113 |
| 11200 | 51 | | 1 | 1 | | CLOSE | PUB00760 841107 |
| 11300 | | | 1 | 1 | | FILE (INFILE); | PUB00770 841113 |
| 11400 | | | | | | | PUB01070 |
| 11500 | 52 | | 1 | 1 | | END LP1431; | PUB01080 850109 |

Figure 10-1 (Part 2 of 2). Use of the ON and SIGNAL Statements

- The ON ENDFILE(SEQFILE) statement establishes the action taken when the end of the input file SEQFILE is reached. The flag MORE_RECORDS is set to NO so that processing of the file will end. Control is then trans-

ferred to the statement immediately following the READ that raised the ENDFILE.

- 2** The ON ERROR statement establishes the action taken when an error is raised during processing of the DIST procedure. If the flag ERR_KEY is equal to YES, which means that the error is raised by the SIGNAL statement, the appropriate error message will be produced. Control then passes to the END_PROC1 label. Any error raised within the on-unit is dealt with by the implicit action, due to the ON ERROR SYSTEM statement.
- 3** Processing begins by setting the initial state of the flags and opening the files.
- 4** A loop is entered to input all the records in file SEQFILE until the ENDFILE condition is raised. The ENDFILE condition causes control to pass to the ON ENDFILE(SEQFILE) statement, and the on-unit is processed. The flag MORE_RECORDS is set on in the on-unit. Then the END statement on statement 31 is carried out. In the next iteration of the loop, the WHILE option will fail, and the loop is exited.
- 5** The input field START_WEEK is checked for valid values. If the value is out of range then the SIGNAL ERROR statement will raise the ERROR condition and the ON ERROR statement at **2** is processed. The flag ERR_KEY is set before the signal is issued to indicate that the error has been raised by a SIGNAL statement.
- 6** This ON ERROR statement establishes the action taken when an error is raised during processing of procedure RANDOM. After the on-unit is processed, control is transferred to the END_PROC1 label to end the DIST procedure. This error action is only active for the RANDOM procedure. When control returns to the DIST procedure, the ERROR on-unit established for DIST is reinstated.
- 7** The ON KEY(INFILE) statement establishes the action to be taken when a KEY condition is raised in the RANDOM procedure. The ON KEY condition is only active for the RANDOM procedure. Control is transferred from this on-unit to the statement immediately following the statement that raised the KEY condition.
- 8** The ONCODE is used to determine the KEY condition raised. If the KEY error is "key not found," then an error message is issued.
- 9** If the raised condition is any KEY condition other than 51, a general key error message is produced.
- 10** Control is transferred here when a valid record is read from file SEQFILE. The file INFILE will be read and processed according to the input fields START_WEEK and NUMB_WEEKS.
- 11** Call an external subroutine to do further processing.

SIGNAL STATEMENT



Chapter 11. Input and Output Statements

This chapter is divided into three sections. The first discusses general input and output statements. The next two sections describe record data transmission, and stream data transmission, by describing the different ways that input and output statements can be used to move different forms of data.

Input and Output

Data is transmitted between storage locations, between external storage mediums, between work stations, and between applications by means of input and output statements. A collection of data external to a program is called a **file**. Transmission of data from a file to a program is called **input**. Transmission of data from a program to a file is called **output**. (“File” can also mean your work station, another program, or a communication line.)

Input and output statements allow a source program to deal with the logical organization of data in a file, rather than with its physical characteristics. To do this, PL/I uses models of AS/400 files. You can write a program without specific knowledge of the input/output devices that will be used when the program runs. A PL/I file can also be connected to different AS/400 files at various times during the running of a program.

There are input and output statements for two types of data transmission: **record** and **stream**.

In **record data transmission**, the file is considered a collection of discrete records. On input, the data is transmitted exactly as it is specified in the record format of the AS/400 file. On output, the data is transmitted exactly as it is specified in the record format specified in the program.

Record data transmission can be used for processing files that are written in any representation, such as binary, decimal, or character.

In **stream data transmission**, the organization of the data into records is ignored, and the data is treated as though it were a continuous stream of individual data values in character form. On input, if necessary, the data is converted from character form in the file to conform with the attributes of the data list item in the program. On output, the data is converted from the attributes of the data list item in the program to character form in the file.

Record data transmission is more versatile than stream data transmission because record transmitted data can be processed in more ways, and record data transmissions can be made to and from more types of AS/400 files. Any type of data can be transmitted using record data transmission because no conversion occurs. However, you must be aware of the structure of the data.

Stream data transmission is more versatile in its formatting of data, but requires more run time.

FILES

The data transmission statements include the following input and output statements:

- GET
- PUT
- READ
- WRITE
- REWRITE
- DELETE.

The OPEN and CLOSE statements are not used for data transmission.

This chapter discusses those aspects of input and output that are common to record and stream data transmission, including files and their attributes, and the relationship of files to AS/400 files. For further information on AS/400 files, see Chapter 6, "AS/400 PL/I File and Record Management" and Chapter 8, "Using AS/400 Files."

Files

A file, as described in an AS/400 PL/I source program, is a model of an AS/400 file. It has significance only within the source program, and does not exist as a physical entity external to the program. However, the program description of the file determines how input and output statements access and process the associated AS/400 files.

You can code the file record descriptions and other file information about the AS/400 file directly into your program (program-described), or you can allow the system to include this file information in your source program (externally described). For a description of how to include file information from externally described files into your source program, refer to "Using the %INCLUDE Directive for External File Descriptions" on page 8-73.

AS/400 Files

The following AS/400 file types are supported by AS/400 PL/I:

- Display file
- Physical data base file
- Logical data base file
- Printer file
- Tape file
- Diskette file
- Inline file
- Data file management (DDM) file.

The following AS/400 file types are supported by the System/38 Environment only:

- Communications file
- BSC file.

Use of the File Attributes

You specify information about a file in a DECLARE statement. A description of the syntax rules of this statement is given in "The DECLARE Statement" on page 12-1. The following sections describe the attributes which are common to both record and stream files. These attributes deal with naming files, and with the types and directions of data transmission used with files.

File Name

A name that represents a file is a **file constant**. Each file must be associated with a file constant. To declare a file constant in a DECLARE statement, you specify the FILE attribute. If the file constant is used as the name of an AS/400 file, it must not be more than ten characters long.

Type Of Data Transmission

You use the RECORD and STREAM attributes in the DECLARE statement to specify the type of data transmission used for the file.

RECORD indicates that the file consists of discrete records, each of which consists of one or more data items in any form. Each record is transmitted as an entity to or from a variable. STREAM indicates that the data of the file is considered a continuous stream of data items in character form, assigned from the stream to variables or from expressions into the stream.

The statements with which a file with the RECORD attribute can be used are the input/output statements OPEN, CLOSE, READ, WRITE, REWRITE, and DELETE, and the ON and SIGNAL statements.

The statements with which a file with the STREAM attribute can be used are the input/output statements OPEN, CLOSE, GET, and PUT, and the ON and SIGNAL statements.

Direction of Data Transmission

You use the INPUT, OUTPUT, and UPDATE attributes to determine the direction of data transmission permitted for a file.

INPUT specifies that data is transmitted from a file to the program. It is valid with any file type except a printer file.

OUTPUT specifies that data is transmitted from the program to create a new file or to extend an existing file. It is valid with any file type except an inline file. OUTPUT and STREAM are not valid with diskette files.

UPDATE can only be used with RECORD. It specifies that the data can be transmitted in either direction. It also allows the insertion of records into an existing file and the altering of other records already in that file. UPDATE is valid only with physical and logical data base files, display, communications, and BSC files.

Opening and Closing Files

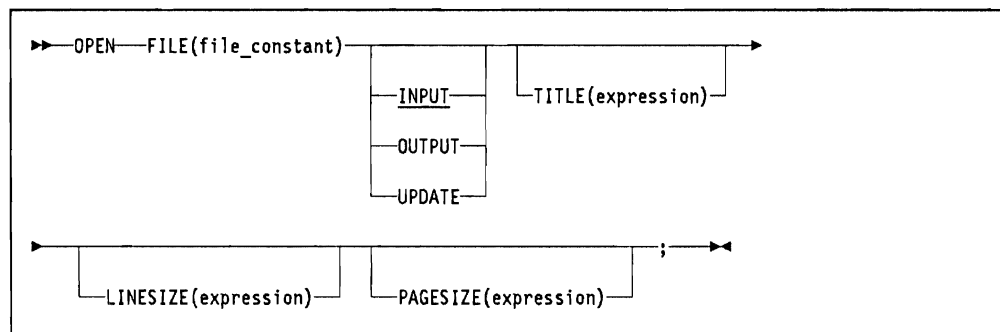
Before a file can be used for input or output, it must be opened. Opening a file involves associating it with an AS/400 file. (If the attempted association is unsuccessful, the UNDEFINEDFILE condition is raised.) When processing is completed, you may close the file. Closing a file involves disassociating the AS/400 file.

PL/I provides two statements, OPEN and CLOSE, to do these functions. Use of these statements, however, is optional.

When a file is opened for SEQUENTIAL INPUT or SEQUENTIAL UPDATE, the current position is the position at the start of the file.

OPEN Statement

The OPEN statement associates a PL/I file with an AS/400 file. For more information on opening files, see "Opening and Closing Files" on page 7-11. The options of the OPEN statement are first evaluated, and the specification of attributes for the file is completed, as necessary. If the association can be made, the file is then associated with the AS/400 file.



FILE(file_constant)

Specifies the name of the PL/I file that is opened. If the file is already open, the evaluated options are not used, and the file is unaffected.

INPUT, OUTPUT, UPDATE

Specifies attributes that augment the attributes specified in the file declaration, with which they must not conflict.

TITLE(expression)

The expression must be a character expression.

The expression specifies the name of the AS/400 file that is opened. It can be a maximum of 33 characters in length. If you code less than 33 characters, it is padded on the right with blanks. If you code more than 33 characters, it is truncated on the right.

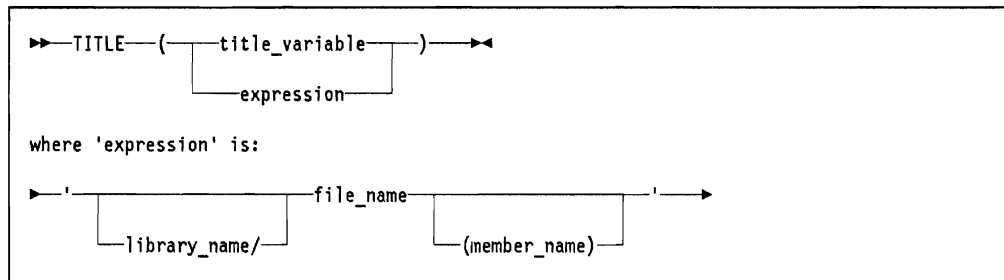
You can use upper and lowercase characters, but the expression is converted to uppercase by the compiler.

An EXTENDED name is delimited by quotes("), which are considered part of the name. The characters can be any of the standard character set except a

blank, an asterisk, a single quote or double quotes. The characters within the quotes will not be monocased.

AS/400 PL/I will only support the EXTENDED name in literals, therefore the only place where they can be used is in the TITLE option of the OPEN statement.

Note: AAA is three characters and "aaa" is five characters. This means that delimited names can be a maximum of eight characters.



title_variable

Can be any valid program variable whose value is a valid AS/400 expression. For example:

```
ACCOUNTS_RECEIVABLE
    = 'ACCTLIB/ACCOUNTS(ACCOUNT1)';
OPEN FILE (ACCTS) UPDATE
    TITLE (ACCOUNTS_RECEIVABLE);
```

expression

Must consist of valid AS/400 file, library and member names.
 If library-name is omitted, *LIBL is assumed.
 If member-name is omitted, the first member in the file is used.

In the following example:

```
OPEN FILE (ACCTS) UPDATE
    TITLE ('ACCTLIB/ACCOUNTS(ACCOUNT1)');
```

ACCOUNTS is the AS/400 file name, ACCTLIB is the AS/400 library name, and ACCOUNT1 is the AS/400 member name.

If you do not specify the library name, your statement would look like this:

```
OPEN FILE (ACCTS) UPDATE
    TITLE ('ACCOUNTS(ACCOUNT1)');
```

LINESIZE(expression)

An integer expression that specifies the length, in characters, of a line during subsequent operations on the file. New lines can be started by means of the control format items or by means of options in a PUT statement. If any attempt is made to position a file past the end of a line, a new line is started, and the file is positioned to the start of this new line.

The implementation-defined values are

| | |
|-------------------|--------|
| Maximum line size | 32 765 |
| Minimum line size | 1 |

OPENING AND CLOSING FILES

The default line size is the record length specified in the AS/400 file.

The LINESIZE option can be specified only for a file that has the STREAM and OUTPUT attributes.

PAGESIZE(expression)

The expression must be an integer expression. It specifies the number of lines on each page. The first attempt to exceed this limit raises the ENDPAGE condition.

The implementation-defined values are:

| | |
|-------------------|--------|
| Maximum page size | 32 767 |
| Minimum page size | 1 |
| Default page size | 60 |

The PAGESIZE option can be specified only for a file that has the STREAM, OUTPUT, and PRINT attributes.

Attribute Merging

To open a file, PL/I uses information about the file from the following sources:

- The attributes specified in a DECLARE statement.
- The attributes specified in the OPEN statement (INPUT, OUTPUT, or UPDATE).
- The options of the ENVIRONMENT attribute.

When a file is opened, the system gets a complete specification of the file by the following procedure:

1. The information from the above sources is collected.
2. The implied attributes are added. (A list of implied attributes is given in "Implied Attributes" on page 12-5.)
3. The default attributes are applied.

If there is any conflict between attributes from any source after the application of default attributes, the UNDEFINEDFILE condition is raised.

Implicit Opening

A file is implicitly opened when one of the data transmission statements listed below is processed for a file for which an OPEN statement has not already been processed.

| <i>Statement</i> | <i>Implied Attributes</i> |
|------------------|---------------------------|
| GET | STREAM INPUT |
| PUT | STREAM OUTPUT |
| READ | RECORD INPUT |
| WRITE | RECORD OUTPUT |
| REWRITE | RECORD UPDATE |
| DELETE | RECORD UPDATE |

An implicit opening caused by one of the above statements is equivalent to preceding the statement with an OPEN statement that specifies the implied attributes.

Examples of File Opening

The following example illustrates attribute merging for an explicit opening:

```
DECLARE LISTING FILE PRINT;
OPEN FILE(LISTING);
```

Because there are no attributes on the OPEN statement, the attribute after merging is PRINT. After applying the implicit attributes, the attributes are STREAM, OUTPUT, and PRINT. After applying the default attributes, the completed attribute set is STREAM, OUTPUT, PRINT, and EXTERNAL.

The following example illustrates implicit opening:

```
DECLARE MASTER FILE KEYED INTERNAL;
READ FILE (MASTER) INTO
    (MASTER_RECORD) KEYTO(MASTERKEY);
```

The attributes after merging (due to the implicit opening caused by processing the READ statement) are KEYED, RECORD, INPUT, and INTERNAL. (No additional attributes are implied.)

The completed attribute set after default application is KEYED, RECORD, INPUT, SEQUENTIAL, ENVIRONMENT(CONSECUTIVE), and INTERNAL.

The following are examples of declarations of file constants, including the ENVIRONMENT attribute:

```
DECLARE FILE#3 FILE INPUT DIRECT
    ENVIRONMENT (BUFSIZE(80)
    CONSECUTIVE);
OPEN FILE(FILE#3);
```

This declaration specifies three file attributes: INPUT, DIRECT, and ENVIRONMENT. DIRECT implies RECORD and KEYED. The scope is external, by default. The ENVIRONMENT attribute specifies that the file is of consecutive organization and contains records 80 bytes long. The KEY option must be specified in each READ statement that refers to this file.

```
DECLARE INVNTY FILE UPDATE
    ENVIRONMENT (KEYDISP(10) KEYLENGTH(5)
    INDEXED);
OPEN FILE(INVNTY);
```

This declaration specifies two file attributes: UPDATE and ENVIRONMENT. The implied attribute is RECORD. SEQUENTIAL and EXTERNAL are the default attributes. INVNTY is a KEYED file with a KEYDISP of 10 and a KEYLENGTH of 5. Although the file actually contains a key within each record, the KEYTO option cannot be specified in a READ statement, because the KEYED attribute was not specified in the DECLARE statement.

RECORD DATA TRANSMISSION

In the above declaration, all necessary attributes are either stated or implied in the DECLARE statement. None of the attributes can be changed in an OPEN statement. This declaration might have been written as follows:

```
DECLARE INVNTY FILE
  ENVIRONMENT(KEYDISP(10) KEYLENGTH(5)
  INDEXED);
```

With such a declaration, INVNTY can be opened for input, output, or update. It could, for example, be opened by means of any of the following statements:

```
OPEN FILE (INVNTY) INPUT;
OPEN FILE (INVNTY) OUTPUT;
OPEN FILE (INVNTY) UPDATE;
OPEN FILE (INVNTY);          /*INPUT BY DEFAULT*/
```

By means of this technique, a file can be opened first as an input file and then closed and later opened as an output file.

CLOSE Statement

The CLOSE statement disassociates the specified file from the AS/400 file with which it was associated by the explicit or implicit file opening.

```
►►—CLOSE—FILE(file_constant);—◄◄
```

FILE(file_constant)

Specifies the name of the PL/I file that is to be disassociated from an AS/400 file.

The CLOSE statement also disassociates from the file all attributes established for it by the implicit or explicit opening process. Although new attributes can be specified for the file constant in a subsequent OPEN statement, all attributes specified for it in a DECLARE statement remain in effect.

Closing a file that is not open has no effect apart from increasing the run time of the program.

A closed file can be opened again either explicitly or implicitly.

If a file is not closed by a CLOSE statement, it is closed at the completion of the run unit. For a complete description of file opening and closing, see "Opening and Closing Files" on page 7-11.

Record Data Transmission

This section describes the file description attributes and input and output statements used in record data transmission. The ENVIRONMENT attribute and details of record data transmission input/output statements for each type of file are described in "The ENVIRONMENT Attribute" on page 7-1. The %INCLUDE directive and details of record data transmission input/output statements for externally described files are described in "Using the %INCLUDE Directive for External File Descriptions" on page 8-73.

In record data transmission, data in a file is considered a collection of records. The READ statement either transmits a single record to a program variable, or sets a pointer to the record in a buffer. The WRITE or REWRITE statement transmits a single record from a program variable to the file. The DELETE statement removes a single record from a file.

Although data can be transmitted to and from a file in blocks, the statements used in record data transmission are concerned only with records. The records are blocked and deblocked automatically. For more information on blocking and deblocking, refer to the description of the BLOCK option of the ENVIRONMENT attribute in "Blocking Option" on page 7-7.

Use of File Description Attributes

You use the file description attribute RECORD to indicate record data transmission. You can also use the file description attributes INPUT, OUTPUT, UPDATE, SEQUENTIAL, DIRECT, KEYED, and ENVIRONMENT to give the system additional information about the file. The syntax rules for all of these attributes are given in "File Attributes" on page 12-6.

AS/400 File Organization

You specify file organization by means of the ENVIRONMENT options CONSECUTIVE, INDEXED, and INTERACTIVE. The organization of a file determines how data is recorded in a file and how the data is subsequently retrieved for transmission to the program.

File organization for each of the types of AS/400 files is discussed in "File Organization Options" on page 7-2.

Other characteristics of AS/400 files can also be specified in the ENVIRONMENT attribute. This attribute is discussed in "The ENVIRONMENT Attribute" on page 7-1.

File Access

The types of file access are:

- SEQUENTIAL
- SEQUENTIAL KEYED
- DIRECT.

You use the SEQUENTIAL, KEYED, and DIRECT attributes to specify these. (The DIRECT attribute implies the KEYED attribute, so when the file access is given as DIRECT, KEYED is understood.) Syntax of these attributes is described in "File Attributes" on page 12-6. They describe how AS/400 files are accessed.

You specify the KEYED attribute when you use one of the key options (KEY, KEYFROM, and KEYTO) of the data transmission statements. For data base files

DATA TRANSMISSION STATEMENTS

with arrival sequence access, or for subfiles, the key in the KEY, KEYFROM, or KEYTO options is a relative record number.

With SEQUENTIAL access, the key options are not valid; with SEQUENTIAL KEYED access, they are optional; with DIRECT access, KEY or KEYFROM is required.

With physical or logical data base files, SEQUENTIAL specifies that records with CONSECUTIVE file organization are accessed using the arrival sequence access path, and that records with INDEXED file organization are accessed using the keyed sequence access path. A file with SEQUENTIAL KEYED access can also be used for direct access or for a mixture of direct and sequential access. Existing records of a file in a sequential update file can be modified or deleted.

Any type of file may have SEQUENTIAL access. Physical and logical data base files, or display files in which subfile processing is used with specific relative record numbers, may have SEQUENTIAL KEYED access. Only physical and logical data base files may have DIRECT access.

Direction of Data Transmission

The attributes INPUT, OUTPUT, and UPDATE are all valid with record data transmission.

INPUT allows the use of the READ statement only. OUTPUT allows the use of the WRITE statement only. UPDATE allows the use of the READ statement with any organization or access, and of the WRITE statement with INTERACTIVE organization or DIRECT access. For the use of the REWRITE and DELETE statements, see "REWRITE Statement" on page 11-15 and "DELETE Statement" on page 11-16. UPDATE is valid only with physical and logical data base files, display, communications, and BSC files.

Variables can be transmitted by record data transmission statements. Although program control data can be transmitted, it may no longer be valid after it is read.

Data Transmission Statements

The statements that transmit records to or from files are READ, WRITE, and REWRITE. The DELETE statement deletes a record from an update file. The attributes of the file determine which statements can be used.

When an input or output statement is processed, the file options are evaluated. If the file is not already open, it is opened implicitly. The validity of the statement is checked against the complete set of attributes. If the statement is valid, then the input or output is processed.

The following sections describe the data transmission statements used for record input/output. The statements are discussed first, followed by the statement options and a discussion of the statements and file attributes that apply to each option.

The file organization and file access that you specify in the DECLARE statement sometimes impose restrictions on the data transmission statements. These restrictions are described in the appropriate sections below.

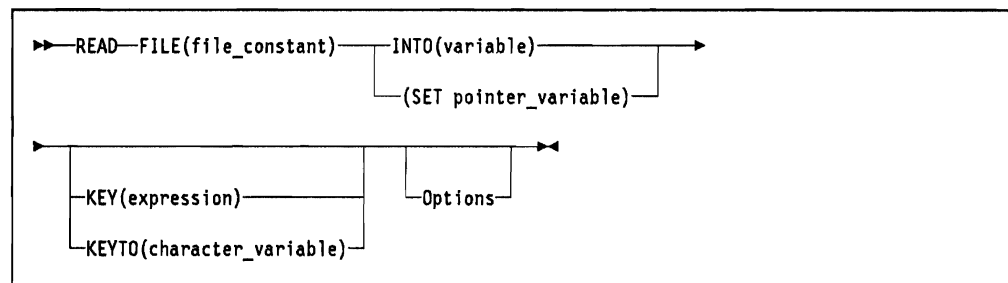
Each data transmission statement contains an OPTIONS option which allows you to access AS/400 system functions. For a discussion of this option and its parameters, see "The OPTIONS Option of Record Data Transmission Statements" on page 7-13.

READ Statement

The READ statement can be used with any RECORD INPUT or UPDATE file. It transmits a record from the file to the program by means of one of the following options:

- INTO, which transfers the record to a variable (move mode).
- SET, which sets a pointer to the record in the buffer (locate mode).

READ a record from a file



FILE(file_constant)

Specifies the name of the PL/I file to which data is to be transmitted. The PL/I file name is associated with an AS/400 file name through the TITLE option of the OPEN statement.

INTO (variable)

Specifies a variable into which the record is read. Variable is a connected aggregate or scalar variable.

SET (pointer_variable)

Specifies a pointer-variable that is set to point to the location in the buffer that contains the record number. Pointer_variable is a simple non-based variable with the pointer attribute.

KEY (expression)

Identifies a particular record. KEY can be CHARACTER VARYING. KEY is not valid with SEQUENTIAL access.

KEYTO (character_variable)

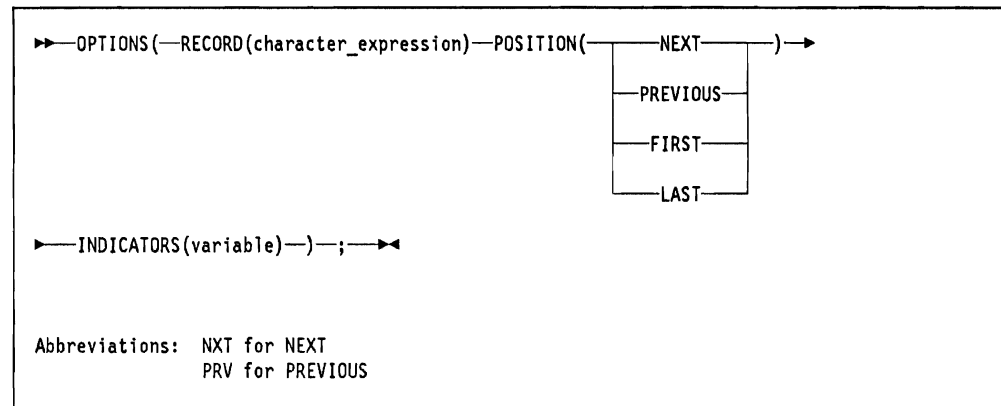
Specifies the variable to which the key of a record will be assigned. KEYTO may be used instead of KEY with SEQUENTIAL KEYED. KEYTO can be CHARACTER VARYING.

DATA TRANSMISSION STATEMENTS

Options

Are different for the three different types of READ statements. The **Options** for each type of READ statement are described below.

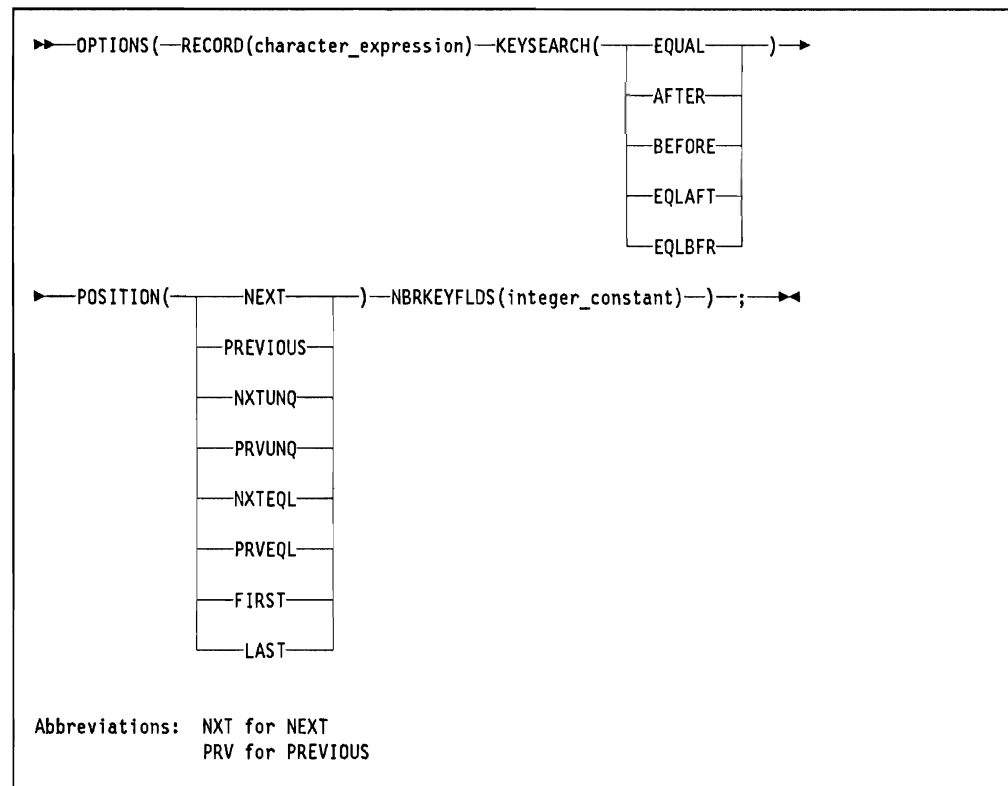
Read from a CONSECUTIVE File



See “The OPTIONS Option of Record Data Transmission Statements” on page 7-13 for a description of the syntax of the **Options** option. Note also that:

- KEY must be a structure reference or scalar expression.
- RECORD, KEY and KEYTO can be CHARACTER VARYING.
- KEY is not valid with SEQUENTIAL access, optional with SEQUENTIAL KEYED access, and required with DIRECT access. KEYTO may be used instead of KEY with SEQUENTIAL KEYED.
- KEY is mutually exclusive with POSITION.
- POSITION is not valid with DIRECT access.
- INDICATORS is valid only with SEQUENTIAL or SEQUENTIAL KEYED access.
- If POSITION is not specified, the default is POSITION(NEXT).

Read from an INDEXED File



See “The OPTIONS Option of Record Data Transmission Statements” on page 7-13 for a description of the syntax of the **Options** option. Note also that:

- KEY must be a structure reference or scalar expression.
- RECORD, KEY and KEYTO can be CHARACTER VARYING.
- KEY is mutually exclusive with POSITION.
- KEY is not valid with SEQUENTIAL access, optional with SEQUENTIAL KEYED access, and required with DIRECT access. KEYTO may be used instead of KEY with SEQUENTIAL KEYED.
- KEYSEARCH is not valid with SEQUENTIAL access.
- KEYSEARCH is not allowed if KEY is not specified.
- POSITION is not valid with DIRECT access.
- If POSITION is not specified, the default is POSITION(NEXT).
- NBRKEYFLDS is not allowed if POSITION(NEXT | PREVIOUS | FIRST | LAST) is specified.

Read from an INTERACTIVE File

```

▶▶—OPTIONS—(—RECORD(character_expression)—▶▶
▶—INDICATORS(variable)—MODIFIED—);▶▶
    
```

See “The OPTIONS Option of Record Data Transmission Statements” on page 7-13 for a description of the syntax of the **Options** option. Note also that:

- KEY must be a structure reference or scalar expression.
- RECORD, KEY and KEYTO can be CHARACTER VARYING.
- KEY is mutually exclusive with MODIFIED.
- KEY is not valid with SEQUENTIAL access and optional with SEQUENTIAL KEYED access. KEYTO may be used instead of KEY with SEQUENTIAL KEYED.
- INDICATORS must be CHARACTER without the VARYING attribute.
- MODIFIED is not valid with SEQUENTIAL access.

WRITE Statement

The WRITE statement can be used with any OUTPUT or INTERACTIVE UPDATE or DIRECT UPDATE file. It transmits a record from the program and adds it to the file.

```

▶▶—WRITE—FILE(file_constant)—FROM(variable)—▶▶
▶—KEYFROM(—expression—)▶▶
▶—OPTIONS(RECORD(character_expression)—INDICATORS(variable))▶▶
▶—;▶▶
    
```

Additional Syntax:

- FROM must be a connected aggregate or scalar variable.
- KEYFROM must be a scalar expression.
- KEYFROM and RECORD can be CHARACTER VARYING.
- KEYFROM is not valid with SEQUENTIAL access, optional with SEQUENTIAL KEYED access and required with DIRECT access.

Note: KEYFROM is required with INTERACTIVE organization and SEQUENTIAL KEYED access.

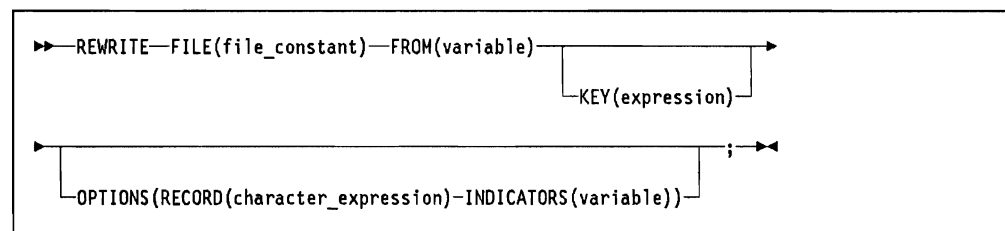
- INDICATORS, when used with CONSECUTIVE organization, is optional with SEQUENTIAL access and not valid with SEQUENTIAL KEYED or DIRECT access.
- INDICATORS is not valid with INDEXED organization.
- INDICATORS must be CHARACTER without the VARYING attribute.

For printer files that contain more than one record format, you must specify the RECORD option. Similarly, you must specify the RECORD option for a WRITE to a multiple format logical file. The only exception is when a format selection program is defined. For information on format selection programs, see the description of the FMTSLR parameter on the CL command CRTLF in the *Programming: Control Language Reference*.

For logical data base files, it is possible to map one record format to multiple base physical file members with the DTAMBRs parameter on the CL commands CRTLF and ADDLFM.

REWRITE Statement

The REWRITE statement replaces a record in an update file. It is only allowed for physical and logical data base files and display files. It cannot be used for a display file with INTERACTIVE organization and SEQUENTIAL access.



Additional Syntax:

- FROM must be a connected aggregate or scalar variable.
- KEY must be a structure reference or scalar expression.
- KEY and RECORD can be CHARACTER VARYING.
- KEY is not valid with SEQUENTIAL access, optional with SEQUENTIAL KEYED access, and required with DIRECT access.
- INDICATORS must be CHARACTER without the VARYING attribute.
- INDICATORS is not valid with CONSECUTIVE organization.

For non-keyed sequential update files, the REWRITE statement must be preceded by a READ statement, with no intervening input or output statements for the same file. It specifies that the last record read from the file is rewritten. Consequently, a record must be read before it can be rewritten. For direct update files and keyed sequential update files, any record can be rewritten regardless of whether it has first been read.

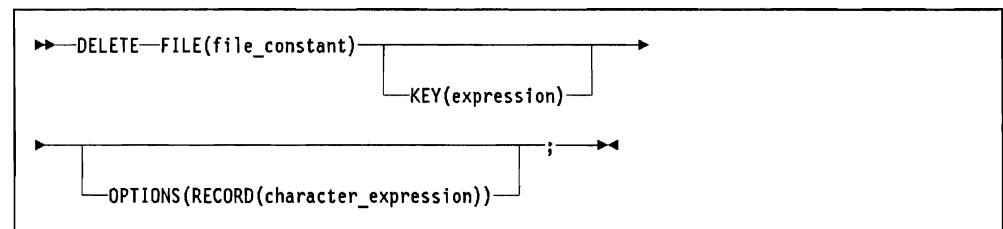
OPTIONS OF DATA TRANSMISSION STATEMENTS

The first statement that accesses the file cannot be a REWRITE statement without the KEY option. No subsequent REWRITE or DELETE statement without a KEY is allowed until another READ statement is processed.

If duplicate keys exist and the key expression locates a duplicate key, the first duplicate in the access path identifies the record rewritten.

DELETE Statement

The DELETE statement deletes a record from an update file. It is only allowed for physical and logical data base files.



Additional Syntax:

- KEY must be a structure reference or scalar expression.
- KEY and RECORD can be CHARACTER VARYING.
- KEY is not valid with SEQUENTIAL access, optional with SEQUENTIAL KEYED access, and required with DIRECT access.

If you omit the KEY option for a sequential update file, the record deleted is the last record read. No subsequent DELETE or REWRITE statement without a KEY is allowed until another READ statement is processed.

If duplicate keys exist and the key expression locates a duplicate key, the first duplicate in the access path identifies the record deleted.

If the first statement that accesses the file is a DELETE statement, it must specify the KEY option.

Options of Record Data Transmission Statements

This section describes the options of record data transmission statements. The options can appear in any order in the statements. Figure 11-1 shows valid combinations of statement options and file organizations. For the complete rules on using these statements, refer to Appendix C, "Valid Combinations of Options for Input/Output Statements" on page C-1.

OPTIONS OF DATA TRANSMISSION STATEMENTS

| Statement Options | CONSECUTIVE Organization | INDEXED Organization | INTERACTIVE Organization (1) |
|-------------------|--------------------------|----------------------|------------------------------|
| READ | | | |
| FILE | R | R | R |
| INTO SET | R | R | R |
| KEY | (2) | (2) | (2) |
| KEYTO | (3) | (3) | (3) |
| OPTIONS | O | O | O |
| RECORD | O | O | O |
| KEYSEARCH | - | O (4) | - |
| POSITION | O (5) | O (5) | - |
| NBRKEYFLDS | - | O | - |
| INDICATORS | O (5) | - | O |
| MODIFIED | - | - | O (4) |
| WRITE | | | |
| FILE | R | R | R |
| FROM | R | R | R |
| KEYFROM | (2) | (2) | (2) |
| OPTIONS | O | O | O |
| RECORD | O | O | O |
| INDICATORS | - (6) | - | O |
| REWRITE | | | (4) |
| FILE | R | R | R |
| FROM | R | R | R |
| KEY | (2) | (2) | O |
| OPTIONS | O | O | O |
| RECORD | O | O | O |
| INDICATORS | - | - | O |
| DELETE | | | |
| FILE | R | R | - |
| KEY | (2) | (2) | - |
| OPTIONS | O | O | - |
| RECORD | O | O | - |

Figure 11-1. Valid Combinations of Record Statement Options

Key:

- R Required
- O Optional
- Error or ignored

Notes:

- (1) DIRECT access is invalid with INTERACTIVE organization.
- (2) Invalid with SEQUENTIAL access, optional with SEQUENTIAL KEYED access, and required with DIRECT access.
- (3) KEYTO may be used instead of KEY with SEQUENTIAL KEYED access.

OPTIONS OF DATA TRANSMISSION STATEMENTS

- (4) Invalid with SEQUENTIAL access.
- (5) Invalid with DIRECT access.
- (6) Optional with SEQUENTIAL access.

The options are discussed in the following order:

FILE
INTO
FROM
SET
KEY
KEYFROM
KEYTO
OPTIONS

FILE (file_constant) Option

The FILE option must appear in every record data transmission statement. It specifies the file to or from which data is transmitted.

If the file specified is not open, it is opened implicitly.

INTO (variable) Option

The INTO option of the READ statement specifies a variable into which the record is read. Either the INTO or the SET option must be used with every READ statement.

The INTO variable can be a VARYING length string.

If the INTO variable is a structure element, it must be connected.

If the INTO variable is shorter than the record length, the record is truncated on the right, and the RECORD condition is raised.

If the INTO variable is equal to the record length, the record is copied to the variable, and no condition is raised. If the INTO variable is CHARACTER VARYING, the string length is set so that it can be referenced by the LENGTH function.

If the INTO variable is larger than the record length, the record is copied to the variable. If the INTO variable is fixed-length, the RECORD condition is raised.

If the INTO variable is CHARACTER VARYING, the string length is set so that it can be referenced by the LENGTH function; no condition is raised. If the CHARACTER VARYING record read in is shorter than the INTO variable, the bytes of the INTO variable not used for the record retain their previous contents; they are not filled with blanks. If the CHARACTER VARYING record read in is longer than the INTO variable, the record is truncated on the right, and the RECORD condition is raised. The byte count at the beginning of the record is changed so that it holds the number of bytes actually read in.

OPTIONS OF DATA TRANSMISSION STATEMENTS

The following example specifies that the next sequential record is read into the variable RECORD1:

```
READ FILE (DETAIL) INTO (RECORD1);
```

FROM (variable) Option

The FROM option must be used in WRITE and REWRITE statements. It specifies the variable from which the record is written.

If the FROM variable is a structure element, it must be connected.

If the FROM variable is shorter than the record length, it is padded on the right with blanks. If the FROM variable is longer than the record length, it is truncated on the right.

In the following example, the WRITE and REWRITE statements specify that the value of the variable DETAILRECORD is written into the file MASTER. The WRITE statement specifies a record added to a SEQUENTIAL OUTPUT file.

```
WRITE FILE(MASTER) FROM(DETAILRECORD);
```

The REWRITE statement specifies that DETAILRECORD is to replace the last record read from a sequential update file.

```
READ FILE(MASTER) INTO(DETAILRECORD);  
/* PROCESSING. NO I/O TO FILE MASTER */  
REWRITE FILE(MASTER) FROM(DETAILRECORD);
```

SET (pointer_variable) Option

The SET option of the READ statement specifies a pointer variable that is set to point to the location in the buffer that contains the required record. Either the INTO or the SET option must be used with every READ statement.

You can use the SET option to avoid raising the RECORD condition. You can also use the SET option to improve program performance: the program will run faster if it does not move the record from the buffer to your own data area.

A READ statement transfers a block of data from the data set to a buffer, if necessary, and then sets a pointer variable named in the SET option to point to the location in the buffer of the next record. The data in the record can then be processed by means of a reference to the based variable associated with the pointer variable.

Alignment errors can occur if the mapping of the based variable does not exactly match the mapping of the input record in the buffer. For example, if the input record contains an unaligned BINARY FIXED (31) field, the based variable must be declared with the UNALIGNED attribute. Otherwise, an error would occur, since the default for BINARY FIXED variables is ALIGNED.

If you specify the SET option, the record condition will not be raised.

OPTIONS OF DATA TRANSMISSION STATEMENTS

The record is available only until the processing of the next input/output operation or CLOSE statement that refers to the same file.

The following example specifies that the value of the pointer variable P is set to the location of the next sequential record in the buffer:

```
READ FILE(X) SET(P);
```

The pointer is invalidated by the next operation on this file.

If indicators are defined within the record buffer, the pointer is set to the start of the indicators, which are located at the start of the record buffer.

KEY (expression) Option

The KEY option of the READ, REWRITE, and DELETE statements identifies a particular record.

The KEY option is required for DIRECT files, and optional for SEQUENTIAL KEYED files.

The KEY expression can be a scalar expression or a structure reference.

If a scalar expression is used in the KEY option, it is evaluated and, if necessary, converted to a binary fixed-point value with a precision of 31 for a CONSECUTIVE or INTERACTIVE file, or to a character value for an INDEXED file. This key determines which record will be processed. If the specified key is not valid, the KEY condition is raised.

For an INDEXED file, the data type of the key should match the data type of the key field(s) defined for the record format. If you are using numeric data base keys, the key you reference must be an element of a structure. Otherwise, the key is converted to character format.

The number of key fields included in the key is determined by the NBRKEYFLDS option (see "NBRKEYFLDS Parameter" on page 7-18). If NBRKEYFLDS is not specified, the length of the expression (or its current length, if it is CHARACTER VARYING) is passed to the system. Refer to the *Programming: Control Language Reference* for information on the generic key search that will be processed if the key length that is passed is between key field lengths.

If there is a duplicate key, the key that is found depends on the KEYSEARCH value. If the value is BFR or EQLBFR, the last duplicate key value is located. For all other values of KEYSEARCH, the first duplicate key value is located.

The following example specifies that the record identified by the value in the variable STKEY is read into the variable ITEM:

```
READ FILE(STOCK) INTO(ITEM) KEY(STKEY);
```

KEYFROM (expression | *) Option

The KEYFROM option of the WRITE statement specifies a key that identifies the location in the file to which the record is transmitted by the WRITE statement.

KEYFROM is required for files with DIRECT access and optional for files with SEQUENTIAL KEYED access. (An exception: KEYFROM is required with INTERACTIVE SEQUENTIAL KEYED.)

KEYFROM(expression)

The expression in the KEYFROM option is evaluated and, if necessary, converted to a binary fixed-point value with a precision of 31 for CONSECUTIVE or INTERACTIVE organization or to a character value for INDEXED organization. This value is used as the key of the record when it is subsequently written. If the specified key is not valid, the KEY condition is raised.

KEYFROM(expression) may only be used for INDEXED files if the ENVIRONMENT options KEYDISP and KEYLENGTH are specified.

KEYFROM(*)

When KEYFROM(*) is specified, the * indicates that the key is imbedded in the associated record, that a keyed operation is being processed and that the key is not moved into the record by PL/I. Since the AS/400 data base supports non-contiguous keys (which may be of any data type), KEYDISP and KEYLENGTH are not applicable because they imply that the key is contiguous.

If you specify the ENVIRONMENT option DESCRIBED for an INDEXED file, you must specify KEYFROM(*).

The KEYFROM(*) option is not allowed for CONSECUTIVE or INTERACTIVE files.

Any KEYFROM value except * for an INDEXED file will be used to update the key field in the record. The embedded key is overwritten.

The following example specifies that the stored value of LOANREC is written as a record in the file LOANS, and that the value of LOANNO is used as the key with which it can subsequently be retrieved:

```
WRITE FILE(LOANS) FROM(LOANREC) KEYFROM(LOANNO);
```

KEYTO (character_variable) Option

The KEYTO option of the READ statement specifies the variable to which the key of a record will be assigned.

KEYTO may be specified instead of the KEY option for a file with SEQUENTIAL KEYED access.

If the KEYTO value is a character scalar, and the file is CONSECUTIVE or INTERACTIVE, the relative record number, which is returned by the system in fixed binary format, is converted to character format. If the KEYTO value is a character scalar and the file is INDEXED, no conversion is attempted. The key

STREAM DATA TRANSMISSION

data returned by the system is moved into the character variable as if the key were character.

For physical or logical data base files for which CONSECUTIVE organization is specified, the relative record number is returned. For subfiles the relative record number within the subfile is returned. For physical or logical data base files for which INDEXED organization is specified, the key value of the record just read is returned.

The following example specifies that the next record in the file DETAIL is read into the variable INVTRY and that the key of the record is made available in the variable KEYFLD:

```
READ FILE(DETAIL) INTO(INVTRY) KEYTO(KEYFLD);
```

OPTIONS Option

The OPTIONS option of the READ, WRITE, REWRITE, and DELETE statements is implementation-defined, and is discussed in "The OPTIONS Option of Record Data Transmission Statements" on page 7-13.

Stream Data Transmission

There are three types of stream data transmission: stream data transmission, keyed data transmission, and record data transmission. Stream data transmission is the slowest form of transmission, and keyed data transmission is the fastest.

This section describes the input and output statements used in stream data transmission. Those features that apply to stream and record data transmission, including files, file attributes, and opening and closing files, are described in Chapter 11, "Input and Output Statements."

Stream input/output is permitted with physical data base, logical data base, diskette (input only), tape, printer, and inline files. It is not permitted with display, BSC, or communications files.

In stream data transmission, a file is treated as a continuous stream of data values in character form, without any delimiters between values. A file created or accessed by stream data transmission is, however, considered to consist of a series of lines of data and has a line size associated with it. A line is generally equivalent to a record in the file, though the line size and record size are not necessarily the same. Stream data transmission can move only problem data, not program control data.

Only edit-directed stream data transmission is allowed. Edit-directed data transmission transmits the values of data items and requires that you specify the format of the values in the stream. The values are recorded externally as a string of characters.

On input, the value of each data item is converted, when necessary, to the attributes of the variable it is being assigned to. On output, the value of each data item is converted to the character representation specified by the associated format item and

placed in the stream in a field whose width may also be specified by the format item.

File Description Attributes

You use the file attribute `STREAM` to indicate stream data transmission. The file description attributes `INPUT`, `OUTPUT`, and `PRINT` can also be used. Information on the syntax of these attributes is in “File Attributes” on page 12-6.

Direction of Data Transmission

The `INPUT` and `OUTPUT` attributes are valid with stream data transmission. `INPUT` allows use of the `GET` statement. `OUTPUT` allows use of the `PUT` statement.

Declaring Print Files

You declare a print file by specifying the `PRINT` attribute. This indicates that you intend to print the file; that is, the data associated with the file will appear on printed pages, although it may first be written on some other device.

The first data byte of each record of a print file is reserved for a carriage control character. When you specify `PRINT`, the system provides first character forms control (FCFC) support. For a description of FCFC, see the *Programming: Data Management Guide*.

You can use the `LINE` option or `PAGE` option of the `PUT` statement only if you specify `PRINT`.

Data Transmission Statements

The same conversion restrictions apply for the `GET` and `PUT` statements as are described in “Assignment Statement” on page 13-1.

The variables to which data items are assigned, and the expressions from which they are transmitted, are generally specified in the **data list** of a **data_specification** in each `GET` or `PUT` statement, which also contains a **format list**. Each data list item in the data list is associated with a data format list item in the format list. A format list item specifies the format of the data item on the external medium. The statements can also include options that specify the position of the data items in the stream relative to the preceding data items.

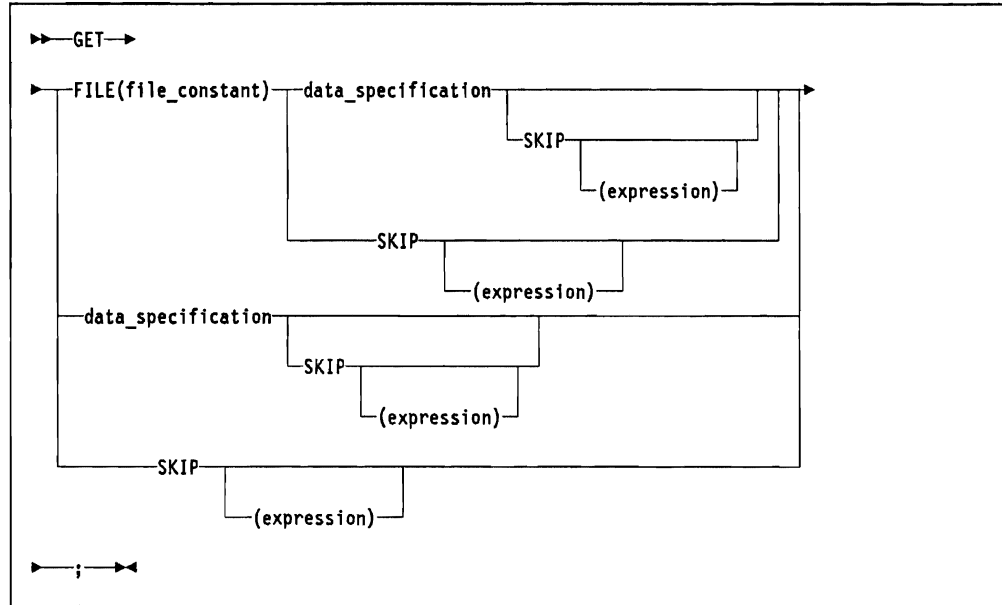
Only stream files can be processed with the `GET` and `PUT` statements.

The following sections describe the `GET` and `PUT` statements, first discussing each statement and then the statement options and the file attributes to which they apply.

DATA TRANSMISSION STATEMENTS

GET Statement

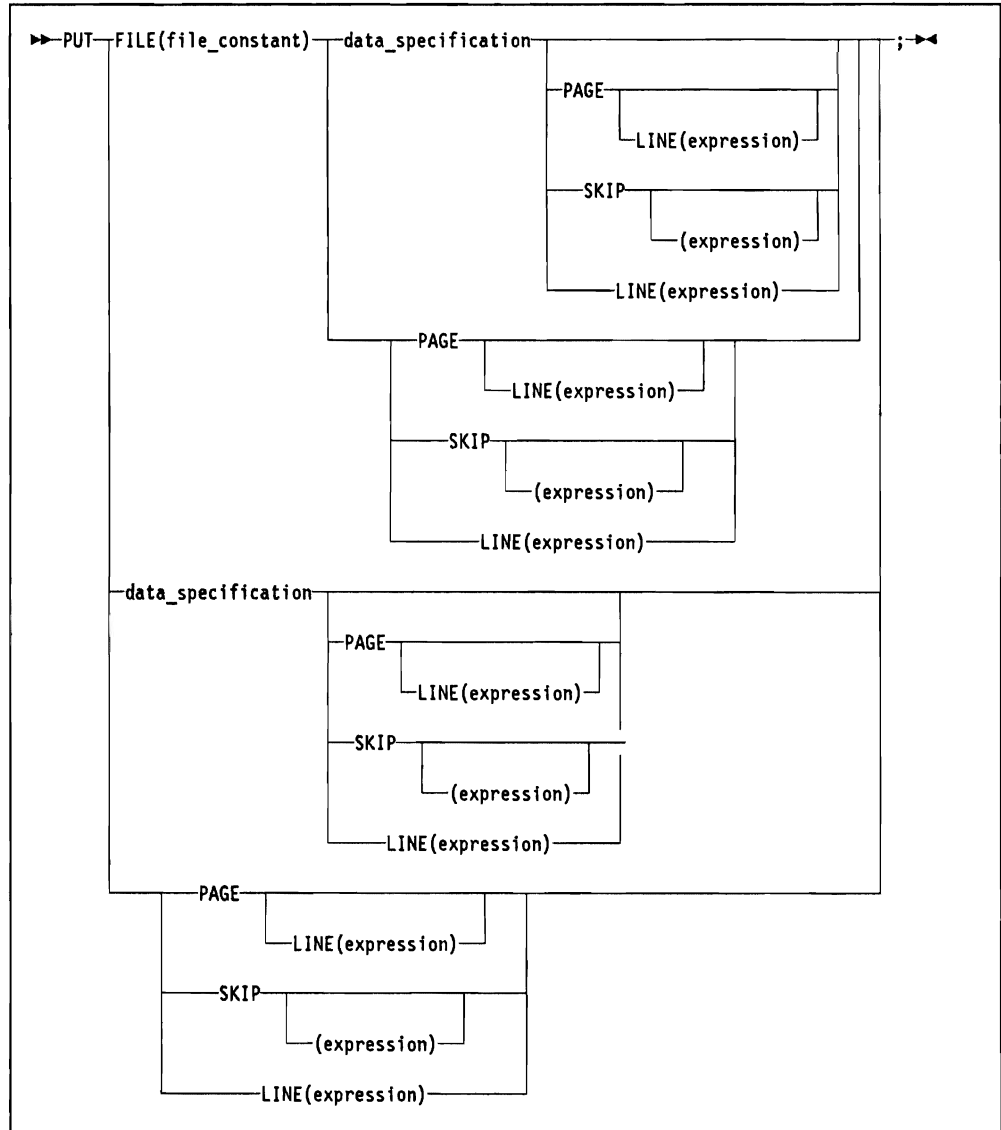
The GET statement is a stream input data transmission statement that assigns data from a file to one or more variables.



You can omit the data specification only if you include the SKIP option.

PUT Statement

The PUT statement is a stream output data transmission statement that transmits data to a file.



You can omit the data specification only if you include one of the control options (PAGE, SKIP, or LINE).

OPTIONS OF STREAM DATA TRANSMISSION STATEMENTS

Options of Stream Data Transmission Statements

The options you can specify on stream data transmission statements are given in the following sections.

FILE (file_constant) Option

The FILE option specifies the file to or from which data is transmitted. It must be a stream file.

file_constant

The name of the file to or from which data is transmitted.

If you omit the FILE option from a GET statement, the file SYSIN is the default. If you omit it from a PUT statement, the file SYSPRINT is the default.

If you do not explicitly open the file, it is implicitly opened for stream data transmission when the first GET or PUT statement is processed for the file.

SKIP (expression) Option

The SKIP option specifies a new current line or record within the file.

expression

An integer expression that specifies the number of the line or record, relative to the current line or record, skipped to. The expression must be less than or equal to 32 767 for input and output files, greater than zero for input files and non-print output files, and greater than or equal to zero for print files. If you omit the expression, the default is 1.

The SKIP option takes effect before the transmission of values defined by the data specification (if present). For example, the following statement writes the values of variables X, Y, and Z to the output file SYSPRINT starting on the third line after the current line:

```
PUT EDIT(X,Y,Z) (F(3),A(2),F(4)) SKIP(3);
```

The effect of the SKIP option is the same as for the SKIP format item (see "SKIP Format Item" on page 11-39).

PAGE Option

You can specify the PAGE option only for print files. It defines a new current page within the file.

The page remains current until a PUT statement with the PAGE option is processed, until a PAGE format item is encountered, or until the implicit action for the ENDPAGE condition is processed. A new page is then defined, the page number is increased by one, and the line count is set to 1.

The PAGE option takes effect before the transmission of any values defined by the data specification (if present). If PAGE and LINE appear in the same PUT statement, the PAGE option is applied first.

LINE (expression) Option

You can specify the LINE option only for print files. It defines a new current line for the file.

expression

An integer expression that specifies the number of the new current line on the current page. The expression must be less than or equal to 32 767. If the expression is less than or equal to zero, LINE(1) is assumed.

The LINE option takes effect before the transmission of any values defined by the data specification (if present). If both the PAGE option and the LINE option appear in the same statement, the PAGE option is applied first. For example:

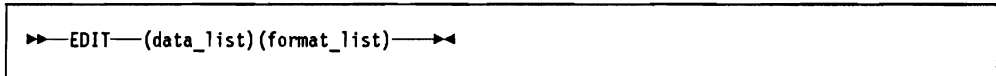
```
PUT FILE(LIST) EDIT(P,Q,R)
    (F(3),A(2),F(4)) LINE(34) PAGE;
```

prints the values of the variables P, Q, and R on a new page, starting at line 34.

The effect of the LINE option is the same as for the LINE format item (see "LINE Format Item" on page 11-39).

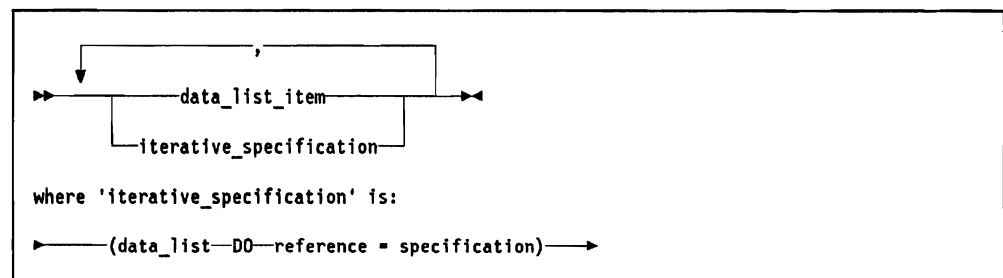
Data Specifications

Data specifications in GET and PUT statements identify the data to be transmitted and its format in the data stream.



Data Lists

The syntax of a data list is shown below:



data_list_item

The data type of a data_list_item must be arithmetic or string.

A data_list_item may be a scalar variable (on input and output), a scalar expression (on output), or an array or structure variable. If you specify an aggregate in a data list, each element of the array or field of the structure is treated as a separate data_list_item, and is paired with a data format item.

DATA SPECIFICATIONS

iterative_specification

The meaning of the specification and of the expressions in the specification are the same as those in a DO statement (described under "DO Statement" on page 13-5).

Each iterative_specification must be enclosed in parentheses. If a data specification contains only an iterative_specification, two sets of outer parentheses are required, since the data list is enclosed in parentheses and the iterative_specification must have a separate set.

When iterative_specifications are nested, the rightmost DO is at the outer level of nesting. For example:

```
GET EDIT (((ARRAY1(I,J)
          DO I = 1 TO 2)
          DO J = 3 TO 4)) (F(3));
```

In this example, there are three sets of parentheses, as well as the set that delimits the subscripts. The outermost set encloses the data list; the next is required by the outer iterative_specification; the third is required by the inner iterative-specification. This statement is equivalent to the following nested do-groups:

```
DO J = 3 TO 4;
  DO I = 1 TO 2;
    GET EDIT (ARRAY1 (I,J)) (F(3));
  END;
END;
```

Values are assigned to the elements of ARRAY1 in the following order:

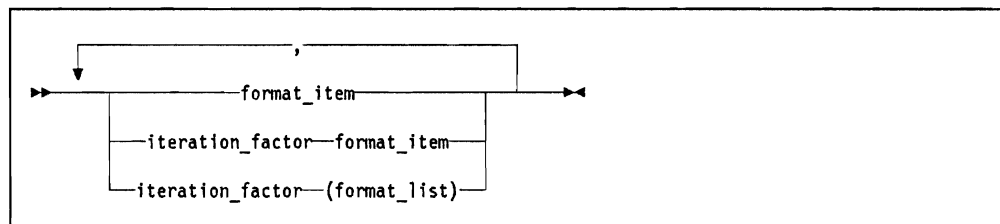
ARRAY1(1,3), ARRAY1(2,3), ARRAY1(1,4), ARRAY1(2,4)

When the specification is completed, processing continues with the next data_list_item.

The maximum level of nesting of iterative_specifications within a data list is 49.

Format Lists

Format lists in GET and PUT statements identify the format of the data on the external medium.



iteration_factor

An integer constant that specifies the number of times the associated format_item or format_list is repeated. A blank must separate the constant and the following format_item.

The associated `format_item` or `format-list` is that item or list of items immediately to the right of the `iteration-factor`.

format_item

Specifies either a data format item or a control format item. The format items and their syntax are shown below. Format items are discussed in “Format Items” on page 11-31.

Format Items and their Syntax

Data Format Items

| | |
|----------------|---|
| character | <code>A[(field_width)]¹</code> |
| bit | <code>B[1 4][(field_width)]¹</code> |
| floating-point | <code>E(field_width[,fractional_digits])</code> |
| fixed-point | <code>F(field_width[,fractional_digits])</code> |

Control Format Items

| | |
|-----------------|---|
| column-position | <code>COLUMN(character_position)</code> |
| line-position | <code>LINE(line_number)</code> |
| paging | <code>PAGE</code> |
| line skipping | <code>SKIP[(relative_line)]</code> |
| spacing | <code>X(field_width)</code> |

Note: ¹`field_width` must be specified in a `GET` statement but is optional in a `PUT` statement.

The first data format item is associated with the first data list item, the second data format item with the second data list item, and so on. If there are fewer data format items than data list items, the format list is reused. If there are more format items than data list items, the excess format items are ignored.

A data format item describes the external format of a single data item.

A control format item specifies the layout of data values within a file.

If one or more control format items are encountered before a data format item, the corresponding control actions are processed first, and then the data list item is transmitted according to the data format item.

In the input stream, all blanks and apostrophes are treated as characters. It is not necessary to enclose strings in apostrophes. Apostrophes should not be doubled, nor should the letter `B` be used to identify bit values. If characters in the stream cannot be interpreted in the manner specified, the conversion condition is raised.

Example of an input specification:

```
GET EDIT (NAME, DATA, SALARY)
      (A(10), X(2), A(6), F(6,2));
```

DATA SPECIFICATIONS

This example specifies the following:

The first ten characters in the stream are considered character data, and are assigned to NAME.

Skip the next two characters.

The next six characters are considered character data, and are assigned to DATA.

The next six characters are considered an optionally signed decimal fixed-point constant and are assigned to SALARY.

In the output stream, blanks are not automatically inserted to separate data values.

String data is left-adjusted within the field width specified; arithmetic data is right-adjusted. Because of the rules for conversion of arithmetic data to character type, which can insert up to three leading blanks (in addition to any blanks that replace leading zeros), at least one blank will precede an arithmetic item in the converted field. However, leading blanks do not appear in the stream unless the specified field width allows for them.

Example of an output specification:

```
PUT EDIT('INVENTORY='||INUM,INVCODE)
      (A,F(5));
```

This example specifies that the character constant 'INVENTORY=' is to be concatenated with the character value of INUM and placed in the stream in a field whose width is the length of the resultant string. Then the value of INVCODE is to be converted to character to represent an optionally signed integer constant placed in the stream right-adjusted in a five-character field (leading characters may be blanks).

Truncation, due to an inadequate field-width specification, is on the left for arithmetic items and on the right for string items. For example:

```
DECLARE INUM      CHARACTER(5),
        INVCODE   FIXED DECIMAL (5);
...
PUT EDIT ('INVENTORY='||INUM,INVCODE)
      (A(13),F(5))
```

This example is similar to the preceding example, except that the length of the character constant 'INVENTORY=' and the length of INUM together exceed the length specified in the format list item A(13). When INUM is concatenated with 'INVENTORY=', its last two characters will be lost.

The PAGE and LINE format items can be used only with print files and, consequently, can only appear in PUT statements. The SKIP, COLUMN, and X-format items can be used with both input and output files.

The PAGE, LINE, and SKIP format items have the same effect as the corresponding options of the PUT statement (and of the GET statement, in the case of SKIP), except that the format items take effect when they are encountered in the format list, whereas the options take effect before any data is transmitted.

The transmission is complete when the last data list item has been processed using its corresponding format item.

Format Items

Format items are specified in a format list in the data specification of a GET or PUT statement. (See "GET Statement" on page 11-24, "PUT Statement" on page 11-25, and "Format Lists" on page 11-28.)

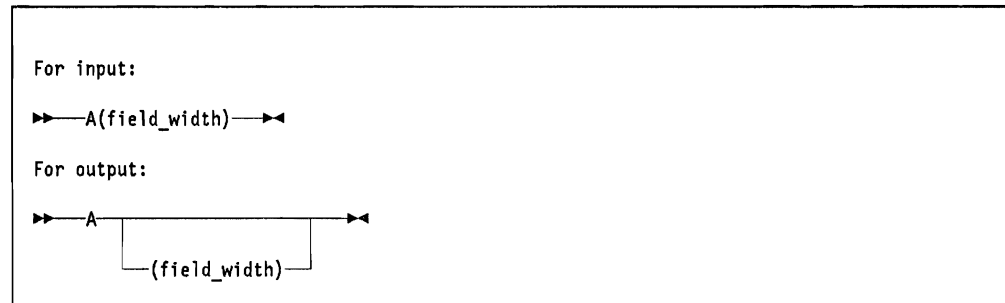
The two types of format items are **data format items**, which describe the external representation of data items, and **control format items**, which specify the formatting of data items.

Control format items take effect before values defined in the data specification are transmitted.

The format items are described in the following sections in alphabetical order.

A-Format Item

The character format item describes the external representation of a string of characters. You can use it for all problem data types.



field_width

An integer constant up to 32 767 that specifies the number of character positions in the data stream that contain, or will contain, the character value.

On input, the specified number of characters is obtained from the data stream and assigned to the data item with any necessary conversion, truncation, or padding. The field_width is always required on input; if it is zero, a null value is obtained. Apostrophes in the stream are treated as characters.

On output, the data list item is converted, if necessary, to a character value and is truncated or extended with blanks on the right to the specified field_width before being placed into the data stream. If the field_width is zero, no characters are placed into the data stream. Enclosing apostrophes are not inserted and contained apostrophes are not doubled. If you do not specify the field_width, the default is equal to the character length of the data list item (after conversion, if necessary, according to the rules given in "Data Conversion" on page 5-27).

For example:

DATA SPECIFICATIONS

```
GET FILE(INFILE) EDIT(ITEM)(A(20));
```

This statement assigns to *ITEM* the next 20 characters in the file, called *INFILE*. The value is converted from its character representation specified by the format item *A(20)* to the representation specified by the attributes declared for *ITEM*.

You can also use the A-format item for input of numeric data containing such characters as currency symbols. For example:

```
DECLARE CHARVARIABLE CHARACTER (5),
        PICVARIABLE PICTURE '$$$$'
                    BASED (POINTER1),
        POINTER1 POINTER;
POINTER1 = ADDR(CHARVARIABLE);
GET EDIT (CHARVARIABLE)(A(5));
```

The *GET* statement causes the next five characters to be assigned to the *CHARVARIABLE*. The associated arithmetic value can then be accessed by means of *PICVARIABLE*.

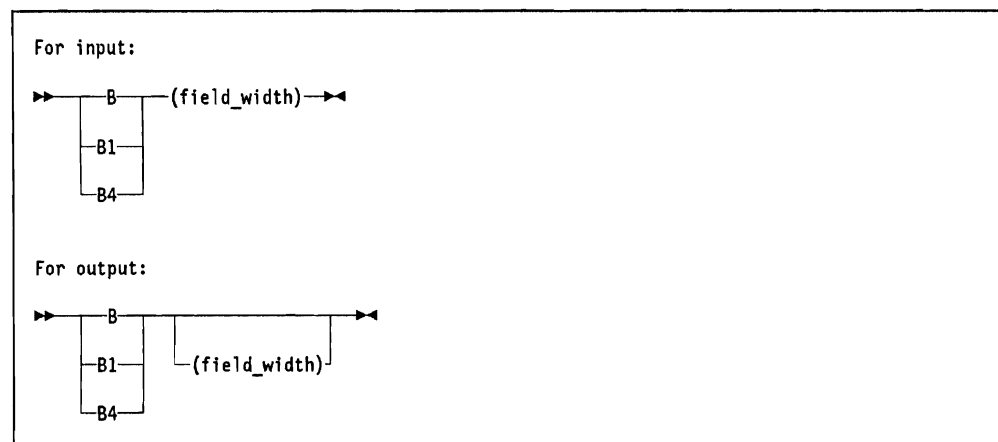
You can also use the A-format item for simple output of any problem data type, in which case the character representations of the data items are written. For example:

```
DECLARE BINVARIABLE BINARY FIXED (15),
        DECVARIABLE DECIMAL FLOAT (10),
        PICVARIABLE PICTURE '999.99';
PUT SKIP EDIT
  (BINVARIABLE,DECVARIABLE,PICVARIABLE)(A,X(5));
```

The *PUT* statement writes the character representations of *BINVARIABLE*, *DECVARIABLE*, and *PICVARIABLE*, separating them by five blanks.

B-, B1-, and B4-Format Items

The bit format item describes a character representation of a bit value. You specify a bit format item as *B*, *B1*, or *B4*, depending on how the bit value is represented.



B, B1

Specify that the value of the data item is represented by the characters 0 and 1.

B4

Specifies that the value of the data item is represented in hexadecimal format.

field_width

An integer constant up to 32 767 that specifies the number of character positions in the data stream that contain, or will contain, the representation of the bit value.

On input, the character representation of the bit value can occur anywhere within the specified field. Blanks, which can appear before and after the bit value in the field, are ignored. Any necessary conversion occurs when the bit value is assigned to the `data_list_item`.

The `field_width` is always required on input; if it is zero, a null value is obtained. Any characters other than those valid for the specified representation (0 or 1 for B or B1 items, or the hexadecimal digits 0 to F for B4 items) raise the conversion condition. For example:

```
DECLARE BITSTRING BIT (16) ALIGNED;
GET EDIT (BITSTRING) (B4(4));
```

If the input stream contains the characters C1C2, the value of BITSTRING after the GET statement is processed is the binary equivalent of hexadecimal C1C2, that is, '1100000111000010'B.

If you use the B4 format item to read data into a character variable, the bit data item is converted to character. For example:

```
DECLARE CHARSTRING CHARACTER(2);
GET EDIT (CHARSTRING) (B4(4));
```

If the input stream contains the characters C1C2, the value of CHARSTRING after processing the GET statement is '11': the first and second bits of the character string C1C2, each of which has a value of one, are each converted to character '1', and the remaining fourteen bits are ignored because CHARSTRING can only contain two characters.

On output, any necessary conversion to bit is processed.

B4 items are assigned to the output field in multiples of four bits. If the converted value of the source is not a multiple of four, it is padded on the right with zeros up to the next multiple of four. For example:

```
DECLARE MASK BIT (25) ALIGNED;
PUT FILE (MASKFILE) EDIT (MASK) (B4);
```

This PUT statement writes the value of MASK to the file MASKFILE as a string of seven hexadecimal characters, by padding MASK with three bits set to zero, producing a string of 28 bits (the next multiple of four).

The character representation of the bit value is left-adjusted in the specified field, and any necessary truncation or extension with blanks occurs on the right. Neither

DATA SPECIFICATIONS

apostrophes nor the identifying letter B (or B1 or B4) are inserted. If the `field_width` is zero, no characters are placed into the data stream. If you do not specify the `field_width`, the default is equal to the length of the data-list-item (after conversion, if necessary, according to the rules given in "Data Conversion" on page 5-27). For example:

```
DECLARE MASK BIT (25) ALIGNED;  
PUT FILE (MASKFILE) EDIT (MASK)(B);
```

This PUT statement writes the value of MASK to the file called MASKFILE as a string of 25 characters consisting of zeros and ones.

COLUMN Format Item

The COLUMN format item positions the file to a specified `character_position` within the current or following line. It can be used with input and output files.

►—COLUMN(`character_position`)—◄

Abbreviation: COL for COLUMN

character_position

An integer constant up to 32 767.

The file is positioned to the specified `character_position` in the current line, provided it has not already passed this position.

If the file is already positioned after the specified `character_position`, the current line is completed and a new line is started; the format item is then applied to this new line.

On input, intervening character positions are ignored. **On output**, they are filled with blanks.

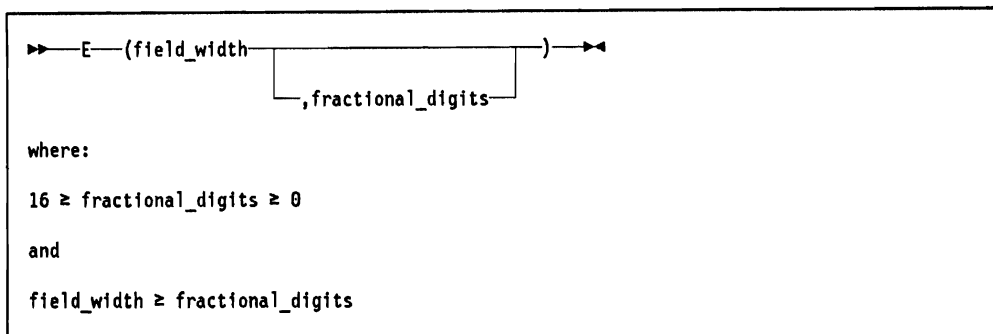
If the specified `character_position` is zero or lies beyond the rightmost `character_position` of the current line, the default `character_position` of 1 is used.

The rightmost `character_position` is determined as follows:

- For output files, it is equal to the line size.
- For input files, it is equal to the length of the current logical record.

E-Format Item

The floating-point format item describes a character representation of a floating-point or fixed-point decimal data item.



field_width

An integer constant up to 32 767. It specifies the total number of characters in the field.

fractional_digits

An integer constant that specifies the number of digits that follow the decimal point. If you omit fractional_digits, it defaults to zero for input. For output, its value is derived from the type of data list item, as follows:

| Data List Item Type | Fractional Digits |
|---------------------|------------------------------------|
| Bit string | $\min(15, \text{ceil}(p1/3.32)-1)$ |
| Character string | 14 |
| Binary fixed | $\text{ceil}(p2/3.32)-1$ |
| Binary float | $\text{ceil}(p3/3.32)-1$ |
| Decimal fixed | $p4-1$ |
| Decimal float | $p5-1$ |
| Picture | $p4-1$ |

Notes

1. $\min(x,y)$ is the smaller of x and y.
2. $\text{ceil}(x)$ is the smallest integer larger than or equal to x.
3. p1 is the length of the bit string.
4. p2 is the number of binary digits that appear in the value of the data list item.
5. p3 is the number of binary digits that are declared or computed to be in the data list item.
6. p4 is the number of decimal digits that appear in the value of the data list item.
7. p5 is the number of decimal digits that are declared or computed to be in the data list item.

On input, the data value in the data stream is the character representation of an optionally signed decimal floating-point or fixed-point constant located anywhere within the specified field. It must conform to the following syntax:

DATA SPECIFICATIONS

`[+|-]significand+`
`[[E|E+|E-|+|-]integer]`

where significand is a decimal fixed-point constant.

Blanks are ignored. They can appear before and after the data value in the field.

The value of `field_width` includes leading and trailing blanks, the exponent position, the positions for the optional plus or minus signs, the position for the optional letter E, and the position for the optional decimal point in the significand.

If no decimal point appears in the significand of the data value, `fractional_digits` specifies the number of character positions in the significand to the right of the assumed decimal point. If a decimal point does appear, it overrides the specification of `fractional_digits`.

On output, the data list item is converted to floating-point and rounded if necessary.

The character string in the output stream has one of the following syntaxes:

- For `fractional_digits = 0 [-]`

digit E

{ + | - }

exponent

The `field_width` must be ≥ 6 for positive values, or ≥ 7 for negative values.

The value zero appears without a sign.

- For `fractional_digits > 0 [-]`

digit.frac_digits E

{ + | - }

exponent

where `frac_digits` is a string of digits of length `fractional_digits`.

The `field_width` must be $\geq 7 + \text{fractional_digits}$ for positive values, or $\geq 8 + \text{fractional_digits}$ for negative values.

The leading digit in the significand is zero only if the value is zero.

The exponent field is three characters long, and can appear in either of two forms, depending on the magnitude of the exponent value. The form `nnb` applies when the exponent value is in the range 0 to 99; two digits are followed by one blank. The form `nnn` applies when the exponent value exceeds 99. You should remember when defining output format that if your exponent is 99 or less, your E-format item will include a blank character following the exponent. Some examples of exponents are `00b`, `99b`, `100`, and `123`.

The conversion from internal decimal fixed-point to character is processed according to the normal rules for conversion. Extra characters may appear as blanks preceding the number in the converted string. And, since leading zeros are converted to

blanks (except for a zero immediately to the left of the decimal point), additional blanks may precede the number. A minus sign will replace one leading blank.

Truncation of data values may occur during conversion (see "Truncation of Floating-Point Data" on page 5-34).

It is an error if the `field_width` is such that the sign or any significant digit is truncated.

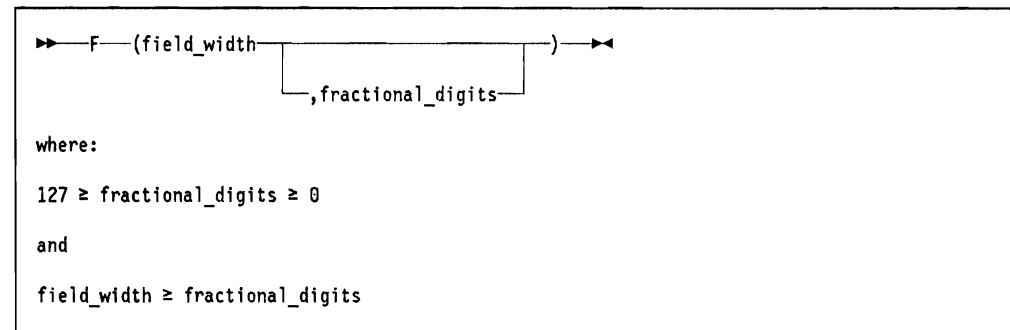
For example:

```
GET FILE(AUDIT) EDIT(COST) (E(10,6));
```

This statement obtains the next ten characters from the file called `AUDIT` and interprets them as a floating-point or fixed-point decimal number. A decimal point is assumed before the rightmost six digits of the significand, but an actual decimal point within the data overrides this assumption. The value of the number is converted to the attributes of `COST` and assigned to this variable.

F-Format Item

The fixed-point format item describes the character representation of a decimal fixed-point arithmetic data item.



field_width

An integer constant up to 32 767. It specifies the total number of characters in the field.

fractional_digits

An integer constant that specifies the number of digits that follow the decimal point. If you omit `fractional_digits`, it defaults to zero.

On input, the data value in the data stream is the character representation of an optionally signed fixed-point decimal constant located anywhere within the specified field. Blanks before and after the data value in the field are ignored. If the entire field is blank, it is interpreted as zero.

The value of `field_width` includes leading and trailing blanks, the position for the optional plus or minus sign, and the position for the optional decimal point.

DATA SPECIFICATIONS

If no decimal point appears in the data value, `fractional_digits` specifies the number of digits in the data item to the right of the assumed decimal point. If a decimal point does appear, it overrides the specification of `fractional_digits`.

On output, the data list item is converted, if necessary, to fixed-point decimal, according to the conversion rules given in “Data Conversion” on page 5-27.

The data value in the stream is the character representation of a fixed-point decimal number, right-adjusted in the specified field.

The `field_width` must be large enough to hold the character representation of the value. The following conditions apply to the `field_width`:

- For `fractional_digits = 0`
$$\text{field_width} \geq 1$$
for positive values, or
$$\text{field_width} \geq 2$$
for negative values.
- For `fractional_digits > 0`
$$\text{field_width} \geq \text{fractional_digits} + 2$$
for positive values, or
$$\text{field_width} \geq \text{fractional_digits} + 3$$
for negative values.

This allows for a decimal point and at least one digit to the left of the decimal point.

If `fractional_digits` is zero, only the integer portion of the number is written; no decimal point appears.

If `fractional_digits` is greater than zero, both the integer and fractional portions of the number are written, and a decimal point is inserted before the rightmost digit position specified by `fractional_digits`. Trailing zeros are supplied when the scale factor is less than the number of fractional digits of the data item after any necessary conversion. If the absolute value of the item is less than 1, a zero precedes the decimal point. Leading zeros are suppressed in all digit positions (except the first) to the left of the decimal point.

Truncation of data values may occur during conversion (see “Truncation of Floating-Point Data” on page 5-34).

The conversion from internal decimal fixed-point to character is done according to the normal rules for conversion. Extra characters may appear as blanks preceding the number in the converted string. And, since leading zeros are converted to blanks (except for a zero immediately to the left of the decimal point), additional blanks may precede the number. A minus sign will replace one leading blank.

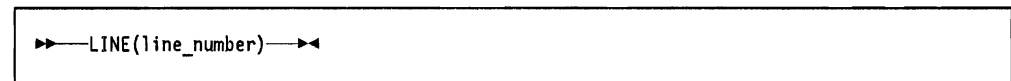
For example:

```
DECLARE TOTAL FIXED DECIMAL (4,2);
PUT EDIT (TOTAL)(F(6,2));
```

The PUT statement specifies that the value of TOTAL is to be converted to the character representation of a decimal fixed-point number and written into the output file SYSPRINT. A decimal point is to be inserted before the last two digits, and the number will be right-adjusted in a field of six characters. Leading zeros will be changed to blanks (except for a zero immediately to the left of the decimal point), and, if necessary, a minus sign will be placed to the left of the first digit.

LINE Format Item

The LINE format item specifies the particular line on the current page of a print file to which the next data list item is transmitted.



line_number

An integer constant up to 32 767.

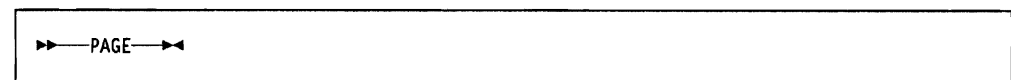
Blank lines are inserted, if necessary, to do the positioning.

The ENDPAGE condition is raised if the specified line_number is:

- Less than the current line_number
- Equal to the current line_number and the current line contains data
- Greater than the limit set by the PAGESIZE option.

PAGE Format Item

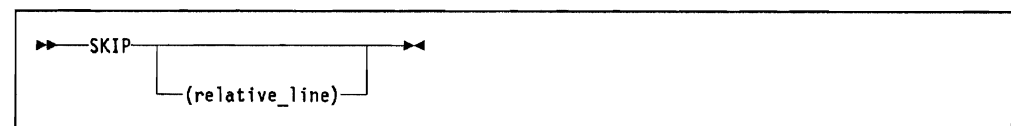
The PAGE format item specifies to start a new page. It can be used only with print files.



Starting a new page positions the file to line 1.

SKIP Format Item

The SKIP format item specifies a new line for a print file, or a new record for a non-print file.



DATA SPECIFICATIONS

relative_line

An integer constant up to 32 767. It specifies the number of the line (relative to the current line) to skip to. `relative_line` must be greater than zero for input files and non-print output files, and greater than or equal to zero for print files. For a print file, `SKIP(0)` means that characters are written on the current line, overwriting characters already written. If you omit `relative_line`, the default is 1.

The file is positioned at the start of the specified line relative to the current line, unless the `ENDPAGE` condition is raised. For print files, the `ENDPAGE` condition is raised if the position of the specified relative line is beyond the limit set by the `PAGESIZE` option of the `OPEN` statement (or by default).

If the `SKIP` format item is the first format item processed after a file has been opened, the file is positioned to the first column of the specified line relative to the first line. For example, `SKIP (0)` would position the file at the first line; `SKIP (1)` would position the file at the second line, and so on.

X-Format Item

The spacing format item controls the spacing of data values in the data stream.

```
▶—X(field_width)—▶
```

field_width

An integer constant up to 32 767. It specifies the number of characters in the data stream between the current position in the stream and the start of the next data field your program will process.

On input, the specified number of characters are bypassed in the data stream and not transmitted to the program. For example:

```
GET EDIT(NUMBER,REBATE) (A(5),X(5),A(5));
```

This statement treats the next 15 characters from the input file, `SYSIN`, as follows: the first five characters are assigned to `NUMBER`, the next five characters are bypassed, and the remaining five characters are assigned to `REBATE`.

On output, the specified number of blank characters are inserted into the stream. For example:

```
PUT FILE(OUT) EDIT(PART,COUNT)
(A(4),X(2),F(5));
```

This statement places, in the file named `OUT`, four characters that represent the value of `PART`, then two blank characters, and finally five characters that represent the decimal fixed-point value of `COUNT`.

Transmission of Array Elements and Structure Fields

If a `data_list_item` is a multi-dimensional array, the array elements are transmitted in row-major order; that is, with the rightmost subscript of the array varying most frequently. Consider the following example:

```
DECLARE SAMPLEARRAY (2,2,2) FIXED BINARY (15);
```

The elements of the array are transmitted in the following order:

```
(1,1,1)
(1,1,2)
(1,2,1)
(1,2,2)
(2,1,1)
(2,1,2)
(2,2,1)
(2,2,2)
```

If a `data_list_item` is a structure, the structure fields are transmitted in the order specified in the structure declaration.

For example:

```
DECLARE 1 ARAY (10),
        2 BFID FIXED DECIMAL(3),
        2 CFID FIXED DECIMAL(3);
.
.
.
PUT FILE(XFIL) EDIT(ARAY) (F(3));
```

produces output ordered as follows:

```
ARAY.BFID(1) ARAY.CFID(1)
ARAY.BFID(2) ARAY.CFID(2)
ARAY.BFID(3) ARAY.CFID(3)
.
.
.
```

However, if the declaration is:

```
DECLARE 1 ARAY,
        2 BFID(10) FIXED DECIMAL(3),
        2 CFID(10) FIXED DECIMAL(3);
.
.
.
PUT FILE(XFIL) EDIT(ARAY) (F(3));
```

the result is ordered as follows:

```
ARAY.BFID(1) ARAY.BFID(2) ... ARAY.BFID(10)
ARAY.CFID(1) ARAY.CFID(2) ... ARAY.CFID(10)
```

If, within a data list, a variable is assigned a value, this new value is used if the variable appears in a later reference in the data list. For example:

PRINT FILES

```
GET EDIT (N,(X(I) DO I=1 TO N))(F(3));
```

When this statement is processed, values are transmitted and assigned as follows:

1. A new value is assigned to N.
2. Elements are assigned to the array X as specified in the iterative specification in the order X(1),X(2),...X(N), with the new value of N being used to specify the number of items assigned.

The following examples show the use of the COLUMN, LINE, PAGE, and SKIP format items in combination with one another:

```
PUT EDIT ('QUARTERLY STATEMENT')
(PAGE, LINE(2), A(19));
PUT EDIT (ACCT#, BOUGHT, SOLD,
PAYMENT, BALANCE)
(SKIP(3), A(6), COLUMN(14),
F(7,2), COLUMN(30), F(7,2),
COLUMN(45), F(7,2),
COLUMN(60), F(7,2));
```

The first PUT statement specifies that the heading QUARTERLY STATEMENT is to be written on line two of a new page in the output file SYSPRINT. The second statement specifies that two lines are skipped (that is, "skip to the third following line") and the value of ACCT# is to be written, beginning at the first character of the fifth line; the value of BOUGHT is to begin at character position 14; the value of SOLD is to begin at character position 30; the value of PAYMENT is to begin at character position 45; and the value of BALANCE is to begin at character position 60.

Print Files

You can control the layout of a print file by means of the options and format items listed in Figure 11-2.

| Statement | Option | Format Item | Effect |
|-----------|----------------------|-------------|------------------------|
| OPEN | LINESIZE(expression) | — | Establishes line width |
| | PAGESIZE(expression) | — | Establishes page depth |

Figure 11-2 (Part 1 of 2). Options and Format Items for Print Files

| Statement | Option | Format Item | Effect |
|-----------|--------------------|---------------------------------------|--|
| PUT | PAGE | PAGE | Skip to new page |
| | LINE(expression) | LINE(<i>n</i>) | Skip to specified line |
| | SKIP[(expression)] | SKIP[(<i>n</i>)] | Skip specified number of lines |
| | — | COLUMN(<i>n</i>) COL(<i>n</i>) | Skip to specified character position in line |
| | — | X(<i>n</i>) | Skip the specified number of characters |

Figure 11-2 (Part 2 of 2). Options and Format Items for Print Files

Note: *n* is an integer constant.

LINESIZE and PAGESIZE establish the dimensions of the printed area of the page, excluding footings. For example:

```

DECLARE REPORT FILE STREAM PRINT,
        N FIXED DECIMAL(3)
        STATIC INITIAL(0);
OPEN FILE(REPORT) PAGESIZE(55)
        LINESIZE(110);
ON ENDPAGE(REPORT)
BEGIN;
PUT FILE(REPORT) SKIP(2) EDIT
        (FOOTING) (A(8));
N = N + 1;
PUT FILE(REPORT) PAGE EDIT
        ('PAGE',N) (A,X(1),F(3));
PUT FILE(REPORT) SKIP (3);
END;
    
```

The OPEN statement opens the file REPORT as a print file. The specification PAGESIZE(55) indicates that each page should contain a maximum of 55 lines, excluding the footing. After 55 lines have been written (or skipped), the next line written or skipped will raise the ENDPAGE condition and process the begin-block. Because the ENDPAGE condition is raised only once for each page, printing continues beyond the specified PAGESIZE.

LINESIZE(110) indicates that each line on the page can contain a maximum of 110 characters. If you attempt to write a line greater than 110 characters, the excess characters are placed on the next line.

The first PUT statement in the begin-block specifies line skipping so that the value of FOOTING, presumably a character value, is to be printed on line 58 (when ENDPAGE is raised, the current line is always PAGESIZE + 1). The page number,

PRINT FILES

N, is incremented, the file is positioned to the next page, and the character constant 'PAGE' is printed together with the new page number. The final PUT statement specifies line skipping so that the file is positioned to line 4. Control returns from the on-unit to the PUT statement that raised the ENDPAGE condition. Any SKIP or LINE option specified in that statement, however, has no further effect.

SYSIN File

The file SYSIN, if not explicitly declared or opened, has the default attributes FILE, STREAM, INPUT, and EXTERNAL, and is associated with the AS/400 file named QINLINE in the AS/400 library named QGPL.

If file SYSIN is explicitly declared with the EXTERNAL attribute, it is associated with the AS/400 file named QINLINE in the AS/400 library named QGPL. If file SYSIN is explicitly declared with the INTERNAL attribute, it is associated with a AS/400 file named SYSIN, which you must create.

You must use SYSIN in a way that is compatible with its use by the compiler. See "Considerations for Opening a Print Stream File" on page 7-12.

SYSPRINT File

The file SYSPRINT is given the attribute PRINT when it is opened, provided that it has the STREAM, OUTPUT, and EXTERNAL attributes. SYSPRINT uses the program name for computer output modules.

A new page is started automatically when the file is opened. If the first PUT statement that refers to the file has the PAGE option, or if the first PUT statement includes a format list with PAGE as the first item, a blank page will appear.

If SYSPRINT is not explicitly declared or opened, it has the default attributes PRINT, FILE, STREAM, OUTPUT, and EXTERNAL, and is associated with the AS/400 file named QPRINT in the AS/400 library named QGPL.

If file SYSPRINT is explicitly declared with the INTERNAL attribute, it is associated with an AS/400 file named SYSPRINT, which you must create.

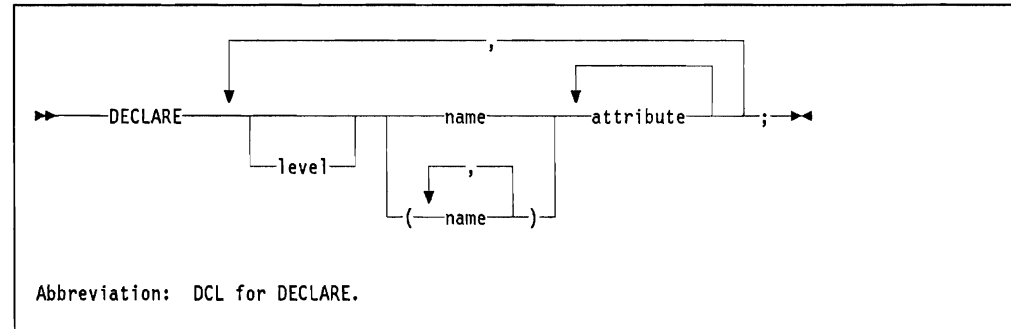
Chapter 12. Declaring Names and Attributes of Variables

This chapter describes the DECLARE statement and its attributes.

The DECLARE Statement

You use the DECLARE statement to explicitly declare the names and attributes of variables. You can declare many variables with a single DECLARE statement.

You can use the DECLARE statement as part of the documentation of your program by specifying all the attributes of a name, even when attributes may be added by default, or by implicit or contextual declaration. You can factor attributes, as described under “Factoring of Attributes” on page 12-2.



level

An integer constant in the range 1 through 255. It specifies the level number of a structure or of a field contained in a structure (see “Structures and Level Numbers” on page 12-39).

name

The name or names being declared. If you want to declare more than one name, see “Factoring of Attributes” on page 12-2.

attribute

All attributes you want to specify for a name must appear in a single DECLARE statement. There can be up to 9999 attributes in a declare statement. You need not declare any default attributes or implied attributes (see “Classification of Attributes” on page 12-2). You can provide the attributes of a variable or a named constant by explicit declaration, by default, or, in the case of built-in function names and built-in subroutines, by context (see “Names” on page 4-12).

A DECLARE statement cannot have a label.

CLASSIFICATION OF ATTRIBUTES

Factoring of Attributes

You can factor attributes common to several names to avoid repeated specification of the same attribute.

To factor attributes, enclose the names of the variables, separated by commas, in parentheses, and follow the parenthesized list by the set of attributes that apply to all of the names. Only identifiers are valid in the parenthesized list.

Factoring cannot be nested; that is, you can only specify one level of parentheses.

Examples of factoring are:

```
DECLARE (A,B,C,D) BINARY FIXED (31);
```

```
DECLARE 1 A, 2(B,C,D) (3,2) BINARY FIXED (15);
```

Classification of Attributes

The attributes are classified by the type of data they represent.

The two types of data in a PL/I program are:

- **Problem data**, which represents values processed by the program. It consists of coded arithmetic, bit, character, and picture data (see "Problem Data Attributes" on page 12-9).
- **Program control data**, which is used to control the processing of the program. It consists of pointer, label, entry, and file data (see "Program Control Data Attributes" on page 12-30).

Figure 12-1 on page 12-3 shows you how to specify attributes for all the types of data.

The rest of this chapter presents the classification, syntax diagrams, and descriptions of the attributes.

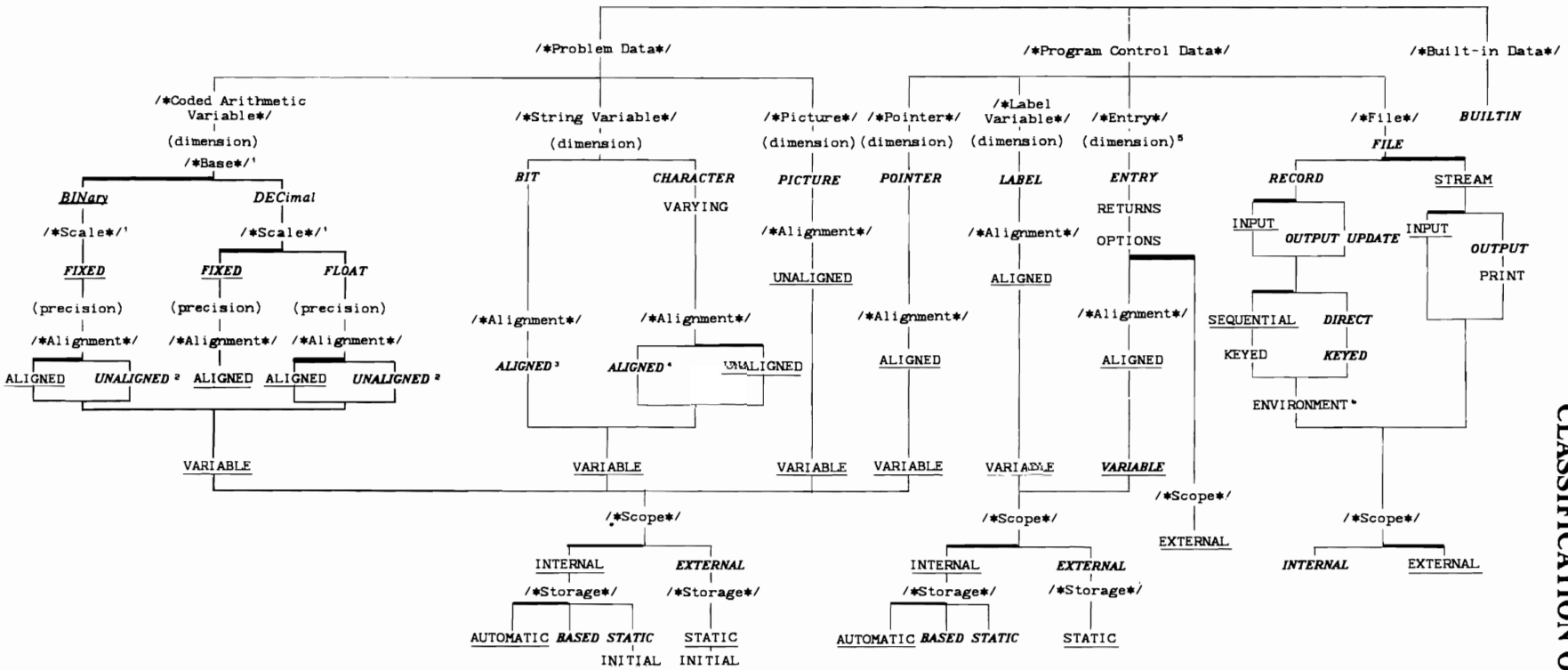


Figure 12-1. Valid Combinations of Attributes

CLASSIFICATION OF ATTRIBUTES

Key

| | |
|---------------|-----------------------------|
| RECORD | must be declared implicitly |
| <u>STREAM</u> | default attribute |
| ———— | default path |
| /* */ | comment |

Footnotes:

¹Only one base or one scale attribute must be specified. The default value then applies for the other.

²UNALIGNED cannot be specified with DECIMAL FIXED.

³ALIGNED must be specified with BIT.

⁴ALIGNED can only be specified if VARYING is also specified.

⁵You cannot specify the dimension attribute for an entry constant.

⁶For more information on the ENVIRONMENT attribute, see "The ENVIRONMENT Attribute" on page 7-1.

Additional Notes:

- Attributes may be specified in any order, except for the following restrictions:
 - The dimension attribute must immediately follow the name or name-list.
 - The precision attribute must immediately follow the base or scale attribute.
- The following restrictions apply to structures:
 - For a major structure name, you can specify any of the scope and storage attributes, except for INITIAL.
 - For a minor structure name, you cannot specify any attributes.
 - For a field name, you can specify only the data and alignment attributes, and the INITIAL attribute.
- The default precisions for coded arithmetic data are shown in "Precision Attribute" on page 12-10. The default string length for string data is 1.
- The implied attributes are listed under "Implied Attributes" on page 12-5.
- Label constants and internal entry constants cannot be declared with a DECLARE statement.
- For structures, INTERNAL and EXTERNAL should be declared in level 1 declarations only. An error message is sent if a member of a structure has a scope specified that is different from the scope specified in the level 1 declaration.

How To Use The Table

Any line through the attributes in this table contains a complete set of the attributes for a name. By proceeding through the table, from top to bottom, you may select a valid combination of attributes for any line.

CLASSIFICATION OF ATTRIBUTES

- Attributes shown in boldface must be declared explicitly (in a DECLARE statement) or implicitly (through the use of another attribute, option or statement). The first attribute in boldface on any line is required if you want any attributes on that line.
- Default attributes are underlined. They are selected for you, on the default path, unless you specify any alternative attributes (shown as branches from the default path).
- All other attributes are optional. You must specify them if you want the declared name to have that particular attribute.
- Exceptions to these points are discussed in the footnotes and additional notes.

Required Attributes

Except for major or minor structure names, you must declare at least one of the following attributes for each name in a DECLARE statement: BINARY, DECIMAL, FIXED, FLOAT, BIT, CHARACTER, PICTURE, POINTER, LABEL, ENTRY, FILE, or BUILTIN.

Implied Attributes

When you declare some attributes, you may imply others. You do not have to declare these attributes, but, just as with default attributes, you can declare them if you wish.

The attributes you specify and the attributes they imply are:

| Explicitly Declared | Implied |
|---------------------------------|--------------------------|
| UPDATE | RECORD |
| SEQUENTIAL | RECORD |
| KEYED | RECORD |
| DIRECT | RECORD, KEYED |
| PRINT | EXTERNAL, OUTPUT, STREAM |
| ENVIRONMENT (KEYDISP KEYLENGTH) | ENVIRONMENT (INDEXED) |

For the file attributes that are implied by implicit opening of a file, see "Implicit Opening" on page 11-6.

CLASSIFICATION OF ATTRIBUTES

File Attributes

A name that represents a file must have the FILE attribute. Such a name is a **file constant**. The characteristics of each file are described with keywords called **file description attributes**.

The file description attributes are either alternative attributes or additive attributes.

An **alternative attribute** is chosen from a group of attributes. If no explicit or implied attribute is given for one of the alternatives in a group, and if one of the alternatives is required, the default attribute is used.

PL/I provides the following three groups of alternative file description attributes (the default attributes are underlined):

RECORD|STREAM
INPUT|OUTPUT|UPDATE
SEQUENTIAL|DIRECT

An **additive attribute** must be stated explicitly or may be implied by another explicitly stated attribute.

The additive attributes are:

KEYED
PRINT
ENVIRONMENT (option_list)

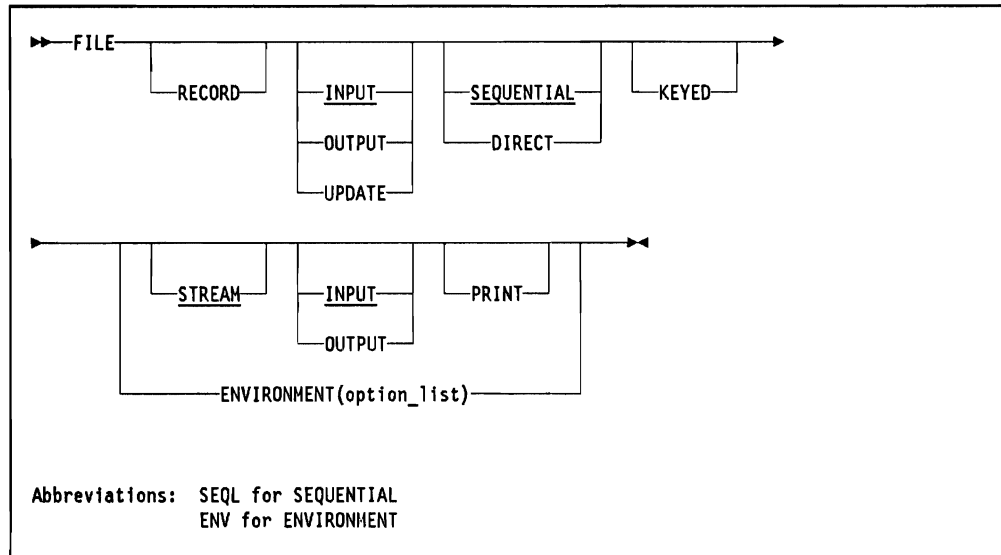
For a list of which file description attributes imply others, see “Implied Attributes” on page 12-5.

Some file description attributes can be specified in an OPEN statement or implied by an implicit opening (see “Implicit Opening” on page 11-6).

You can also specify scope and storage attributes (excluding INITIAL), but the default scope value is EXTERNAL. You cannot specify VARIABLE or include a file constant in an aggregate (see Figure 12-1 on page 12-3).

The syntax of the FILE attribute and the file description attributes is:

CLASSIFICATION OF ATTRIBUTES



FILE Attribute

The FILE attribute indicates that the associated name is a file constant.

Each file name must be explicitly declared with the FILE attribute.

RECORD and STREAM Attributes

The RECORD and STREAM attributes specify the type of data transmission used for the file.

The UPDATE, SEQUENTIAL, DIRECT, KEYED, and ENVIRONMENT attributes can only be used with the RECORD attribute.

The PRINT attribute can only be used with the STREAM attribute.

INPUT, OUTPUT, and UPDATE Attributes

The INPUT, OUTPUT, and UPDATE attributes determine the direction of data transmission permitted for a file. UPDATE must not be used with STREAM.

SEQUENTIAL and DIRECT Attributes

The SEQUENTIAL and DIRECT attributes describe how the records in the file are accessed.

The SEQUENTIAL and DIRECT attributes apply only to a file with the RECORD attribute.

KEYED Attribute

The KEYED attribute indicates that records in the file can be accessed by means of one of the key options (KEY, KEYTO, or KEYFROM) of data transmission statements or of the DELETE statement.

The KEYED attribute applies only to a file with the RECORD attribute.

PRINT Attribute

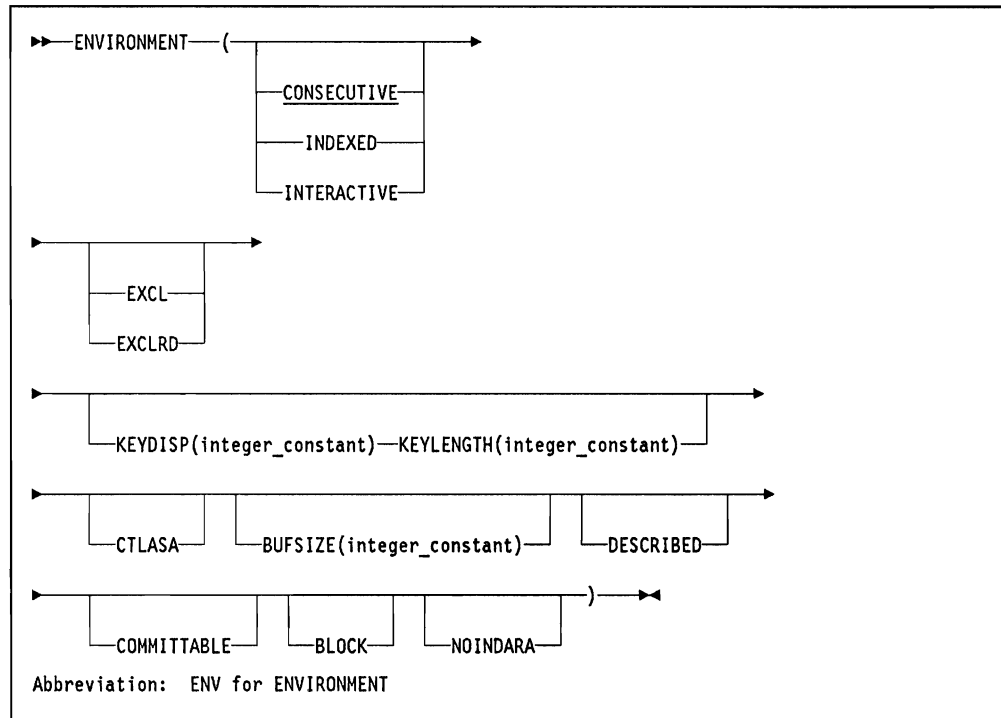
The PRINT attribute indicates that the file is intended for printing; that is, the data associated with the file is to appear on printed pages, although it may first be written on some other device.

DATA TYPES

The PRINT attribute applies only to files with the STREAM and OUTPUT attributes.

ENVIRONMENT Attribute

The options list of the ENVIRONMENT attribute is implementation-defined. It specifies file characteristics that are not part of the PL/I language.



Options in the option-list are separated by blanks or commas. The options are described in “The ENVIRONMENT Attribute” on page 7-1 and summarized in Figure 7-2 on page 7-9.

ENVIRONMENT(CONSECUTIVE) is the default value if you do not specify the ENVIRONMENT attribute.

Do not specify the ENVIRONMENT attribute and the STREAM attribute for the same file.

For a discussion of using file data and the file description attributes for record input/output, see “Use of File Description Attributes” on page 11-9. For a discussion of using file data and the file description attributes for stream input/output, see “File Description Attributes” on page 11-23.

Data Types

The types of data in a PL/I program are:

- **Problem data**, which represents values processed by the program. It consists of coded arithmetic, string, and picture data (see “Problem Data Attributes” on page 12-9).

- **Program control data**, which is used to control the processing of the program. It consists of pointer, label, entry, and file data (see “Program Control Data Attributes” on page 12-30).

For example, the statement:

```
AREA = RADIUS**2 * 3.1416;
```

contains problem data variables and constants. AREA and RADIUS are variables, and the numbers 2 and 3.1416 are constants.

If you want to use the number 3.1416 in more than one place in a program, it may be more convenient to represent it as a variable to which you assign the value 3.1416. Therefore, the above statement could be written as:

```
PI = 3.1416;
AREA = RADIUS**2 * PI;
```

In the last statement, only the number 2 is a constant.

The following program segment contains a **program control data constant** or **named constant** called LOOP:

```
SAMPLE: PROCEDURE OPTIONS (MAIN);
        DECLARE (ITEM1,ITEM2)   FIXED DECIMAL (2);
        DECLARE ITEM3           FIXED DECIMAL (3);
        DECLARE EOF             BIT (1) ALIGNED
                                INITIAL ('0'B) STATIC;
        ON ENDFILE (SYSIN) EOF = '1'B;
        GET EDIT (ITEM1,ITEM2) (COL(1),F(2),F(2));
LOOP: DO WHILE (EOF = '0');
        ITEM3 = ITEM1 + ITEM2;
        PUT EDIT (ITEM3) (COLUMN(10),F(3));
        GET EDIT (ITEM1,ITEM2) (COLUMN(1),F(2),F(2));
        END LOOP;
        END SAMPLE;
```

The name LOOP is declared as a label constant by its appearance as a label prefix. The value of the constant identifies the labeled statement, within a particular activation of the block containing this statement.

Problem Data Attributes

Problem data consists of coded arithmetic, string, and picture data.

VARIABLE can be specified for all problem data. It is the default value. INTERNAL is the default scope value.

Problem data can always be grouped into aggregates by specifying a dimension or by using structures.

PROBLEM DATA ATTRIBUTES

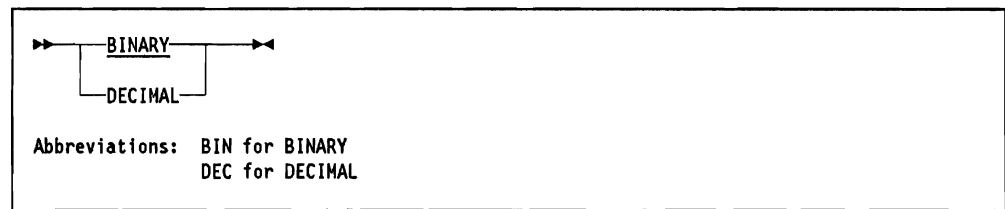
Coded Arithmetic Data Attributes

The types of coded arithmetic data attributes are base, scale, and precision.

Each coded arithmetic variable must be declared with at least a base or a scale attribute. You may also specify precision, UNALIGNED (except with DECIMAL FIXED), VARIABLE, scope and storage attributes, but default values apply if you do not specify any of these. Default values are shown in Figure 12-1 on page 12-3. You can always group coded arithmetic variables into aggregates.

Base Attributes

The base of an arithmetic data item is either binary or decimal.



Scale Attributes

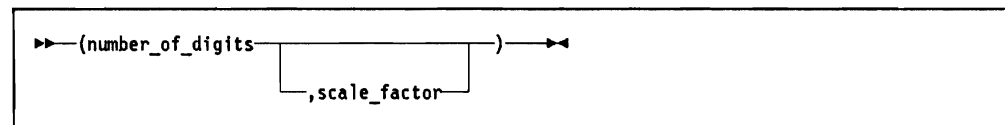
The scale of an arithmetic data item is either fixed-point or floating-point.



Precision Attribute

The precision of a fixed-point data item is the number of digits the data item can have. The precision of a floating-point data item is the minimum number of significant digits (excluding the exponent). For decimal fixed-point data, the precision attribute may include the scale factor (the assumed position of the decimal point, relative to the rightmost digit of the number).

The syntax for fixed-point precision is:



The syntax for floating-point precision is:



number_of_digits

An integer constant that specifies the number of digits (both integers and fractional digits) maintained for data items assigned to the variable. For BINARY variables, it specifies the number of bits. For DECIMAL variables, it specifies the number of decimal digits.

scale_factor

An integer constant that specifies the number of fractional digits for fixed-point data.

For decimal fixed-point data, the scale factor must be in the range 0 through 15. The scale factor must not exceed the number of digits. If you specify a scale factor for binary fixed-point data, it must be 0.

The precision attribute must immediately follow the base or scale attribute.

The maximum number of digits allowed and the default precisions for each data type are shown in the table below.

| Data type | Maximum digits | Default precision |
|------------------|----------------|-------------------|
| DECIMAL FIXED | 15 | (5,0) |
| BINARY FIXED | 31 | (15) |
| DECIMAL FLOAT | 16 | (7) |
| BINARY FLOAT | 53 | (24) |

The maximum length restrictions hold for declared precisions and for the precision of a constant.

The precision attribute is often represented as (p) or (p,q) , where p represents the number of digits, q represents the scale factor, and p is greater than or equal to q .

Decimal Fixed-Point Data

A **decimal fixed-point value** is a rational number, regarded as a sequence of decimal digits with an assumed position of the decimal point.

A **decimal fixed-point constant** consists of one or more decimal digits with an optional decimal point. If you omit the decimal point, it is assumed to be immediately to the right of the rightmost digit. The precision of a decimal fixed-point constant is (p,q) , where p is the total number of digits in the constant and q is the number of digits to the right of the decimal point; $q = 0$ if there is no decimal point. q must be less than or equal to p .

Examples of decimal fixed-point constants and their precisions are:

PROBLEM DATA ATTRIBUTES

| Constant | Precision |
|----------|-----------|
|----------|-----------|

| | |
|--------|-------|
| 3.1416 | (5,4) |
| 455.3 | (4,1) |
| 732 | (3,0) |
| 003 | (3,0) |
| 5280 | (4,0) |
| .0012 | (4,4) |

You declare a decimal fixed-point variable with the DECIMAL, FIXED, and precision attributes. A variable declared as DECIMAL FIXED (p,q) can hold values represented by p decimal digits, q of which are to the right of the assumed decimal point. For example:

```
DECLARE VARIABLE1 FIXED DECIMAL (5,4);
```

```
DECLARE VARIABLE2 FIXED DECIMAL (3,0);
```

These DECLARE statements specify that VARIABLE1 and VARIABLE2 are decimal fixed-point variables with values in the range -9.9999 through +9.9999, and -999 through +999 respectively. A value assigned to VARIABLE1 or VARIABLE2 is converted to decimal fixed-point and aligned on the decimal point.

If the value 1.22229 is assigned to VARIABLE1 and VARIABLE2, the resulting value of VARIABLE1 is 1.2222 and the resulting value of VARIABLE2 is 001. If the value assigned is 123.999, the resulting value of VARIABLE1 is undefined because it is too large; the FIXEDOVERFLOW condition will also be raised. The resulting value of VARIABLE2 is 123.

Decimal fixed-point data is represented in storage as packed decimal. Packed decimal data is stored two digits to the byte. The rightmost byte holds only one digit; its rightmost four bits hold the sign indication. Consequently, a decimal fixed-point data item with an even number of digits contains zeros in the leftmost four bits of the leftmost byte.

Binary Fixed-Point Data

A **binary fixed-point value** is an integer, regarded as a sequence of binary digits. (Binary fixed-point data always has a scale factor of zero.)

There are no binary fixed-point constants.

You declare a binary fixed-point variable with the BINARY, FIXED, and precision attributes. A variable declared as BINARY FIXED (p) can hold integers represented by p binary digits. For example:

```
DECLARE FACTOR BINARY FIXED (20);
```

In this example, FACTOR is to represent binary fixed point data items with a precision of 20 binary digits.

A binary fixed-point data item requires either a halfword or word of storage. A halfword contains 15 binary digits plus a sign bit, and a word (sometimes called a fullword) contains 31 binary digits plus a sign bit. Any binary fixed-point data item with a precision of 15 or less is stored as a halfword. Any binary fixed-point data item with a precision greater than 15, up to the maximum precision of 31, is stored as a fullword. The maximum value of a binary fixed-point data item is $2^{15} - 1$ (or 32 767) for the halfword form, and $2^{31} - 1$ (or 214483647) for the fullword form. Negative values are stored in "two's complement notation." The declared number of digits begins from the low-order positions.

Decimal Floating-Point Data

A decimal floating-point data item is an approximation of a real number, and consists of a sign, a significand, and an exponent. Its value is the signed product of (a) its significand, and (b) 10 raised to the power of its exponent.

A decimal floating-point constant consists of a decimal fixed-point constant (that corresponds to the significand), followed by the letter E, followed by an optionally signed decimal integer constant (that corresponds to the exponent). The precision of a decimal floating-point constant is taken to be the number of digits in the significand. Examples of decimal floating-point constants and their precisions are:

| Constant | Precision |
|--------------|-----------|
| 15E-23 | (2) |
| 15E23 | (2) |
| 4E-3 | (1) |
| 0.4E-2 | (2) |
| 1.96E + 07 | (3) |
| 438E0 | (3) |
| 3141593E-6 | (7) |
| .003141593E3 | (9) |

The last two examples represent the same value.

You declare a decimal floating-point variable with the DECIMAL, FLOAT, and precision attributes. A variable declared as DECIMAL FLOAT (p) can represent real numbers with a precision of p significant decimal digits. For example:

```
DECLARE LIGHTYEARS DECIMAL FLOAT (5);
```

In this example, LIGHTYEARS is to represent decimal floating-point data items with a precision of 5 significant decimal digits.

The maximum precision allowed for decimal floating-point variables is 16; the default precision is 7.

If you declare a floating-point variable with a precision less than or equal to 7, the value is represented internally in short format. If greater than 7 and less than or equal to 16, the value is represented internally in long format. Short and long floating point formats are described below.

PROBLEM DATA ATTRIBUTES

Binary Floating-Point Data

A **binary floating-point data item** is an approximation of a real number, and consists of a sign, a significand, and an exponent. Its value is the signed product of (a) its significand, and (b) 2 raised to the power of its exponent.

There are no binary floating-point constants.

You declare a binary floating-point variable with the **BINARY**, **FLOAT**, and **precision** attributes. A variable declared as **BINARY FLOAT (p)** can represent real numbers with a precision of **p** significant **binary** digits. In the following example,

```
DECLARE TRIALNO FLOAT (16);
```

TRIALNO represents a binary floating-point data item with a precision of 16 binary digits in the significand.

Note: **BINARY** is the default attribute, if you do not specify a base attribute.

The maximum precision allowed for binary floating-point data items is 53; the default precision is 24.

If you declare a floating-point variable with a precision less than or equal to 24, the value is represented internally in short format; if the precision is greater than 24 and less than or equal to 53, the value is represented internally in long format. Short and long floating-point formats are described below.

Internal Representation of Floating-Point Data

A floating-point number is represented in the AS/400 storage as a bit string consisting of three parts. The left part represents the sign of the number, the middle part represents the exponent of the number, and the right part represents the significand of the number. Its value relative to a binary base can be expressed in the form:

$$(\text{sign})(\text{significand}) \cdot (2)^{(\text{exponent})}$$

where ***** and ****** denote the multiplication operator and the exponentiation operator respectively.

The representation of a floating-point number can be in either of two formats. The short format stores in four bytes any number whose magnitude is within the range of (2^{*-126}) to $((2^{-23}) \cdot 2^{**127})$, or approximately 10^{*-38} to 10^{**38} . The long format stores in eight bytes any number whose magnitude is within the range of (2^{*-1022}) to $((2^{-52}) \cdot 2^{**1023})$, or approximately 10^{*-308} to 10^{**308} . The signed zero number can also be stored using both formats.

The sign of the number is represented by a single bit. A zero indicates a positive sign, and a one indicates a negative sign.

The exponent of the number is represented by eight bits if short format or eleven bits if long format. This set of bits is treated as a binary integer. The exponent value is adjusted prior to being stored by adding 127 if short format or 1023 if long

format. This means that the sign of the exponent does not have to be stored. This means that the exponent value is in the range 1 to 254 if short format, or 1 to 2046 if long format and it can be treated as unsigned. A stored exponent value of 0 serves to identify the signed zero value. A stored exponent value of 255 for short format or 2047 for long format, together with certain significand values, serves to identify two symbolic number values, signed infinity and "not a number (NaN)."

The significand of the number is represented by 23 bits if short format or 52 bits if long format. This set of bits is treated as a binary integer. In the case of the signed zero value, the significand is zero.

The supported ranges of floating-point numbers are applicable when a floating-point number is stored in normalized form. A floating-point number is always stored in normalized form when an operation that produces a floating-point result is successfully processed. A floating-point result is normalized just prior to being stored. Normalization involves shifting the significand to the left while decrementing the exponent until the leading significand bit becomes one; this leading one bit is then dropped before storing the significand, since its presence is implied.

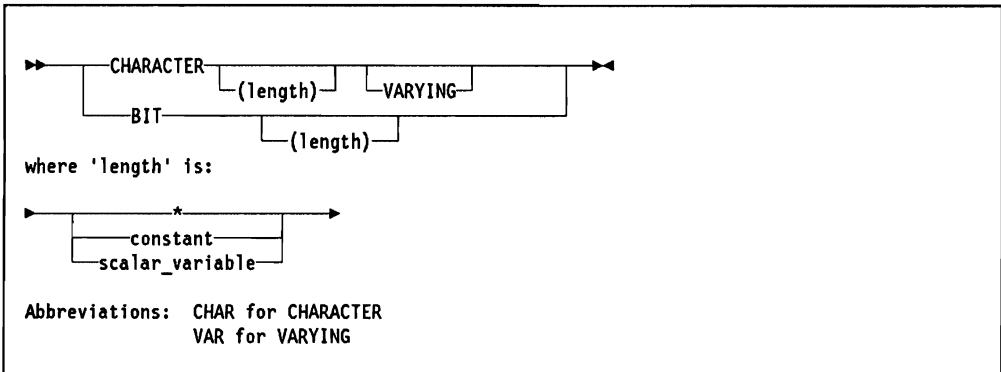
After a floating-point number has been normalized, the exponent value is checked to determine if it is outside the range allowed in the format of the result field: if it is below the minimum limit, a floating-point underflow condition is raised; if it is above the maximum limit, a floating-point overflow condition is raised.

String Data Attributes

String data attributes refer to either character data or bit data. A string is a contiguous sequence of characters or bits that is treated as a single data item.

You must specify BIT or CHARACTER. You can use the VARYING attribute to indicate varying-length character strings. You cannot specify VARYING for bit strings. You may also specify alignment (with the restrictions shown in Figure 12-1 on page 12-3), VARIABLE, scope and storage attributes. Default values for these are shown in Figure 12-1 on page 12-3. String data can be grouped into aggregates.

The syntax of the string data attributes is:



BIT and CHARACTER Attributes

The BIT attribute declares a variable that can hold bit values of a given length. The CHARACTER attribute declares a variable that can hold character values of a given length.

length

Specifies the length of a string. The default length is 1. The maximum length of a bit or nonvarying character variable is 32 767; the maximum length of a VARYING character variable is 32 765. The minimum length of a bit or character variable is zero. Specify the length of a bit variable by number of bits, and the length of a character variable by number of characters.

Specify the length attribute as follows:

- If the variable is static, based, or a member of a structure, the length must be an integer constant.
- If the variable is automatic, the length must be an integer constant or a non-based scalar variable. If the variable is automatic, you must ensure that storage is currently allocated to it at the time you declare the bit or character string using the variable to specify the length of the string.
- If the variable is a parameter in an internal procedure, or a parameter descriptor in an ENTRY declaration, the length must be an integer constant or an asterisk. The asterisk indicates that the parameter will have the length of the corresponding argument passed by the calling procedure.

All bit variables must have the ALIGNED attribute. If you do not specify ALIGNED, the compiler assumes ALIGNED and generates a warning message.

Bit Data

A **bit value** is a sequence of binary digits stored in consecutive bits. The storage for a declared bit variable, including each element of a bit array, always starts at a byte boundary.

A **bit constant** is either a series of binary digits enclosed in apostrophes and followed immediately by B or B1, or a series of hexadecimal digits enclosed in apostrophes and followed immediately by B4. The value of a B or B1 constant is the string consisting of the binary digits between the enclosing apostrophes. The value of a B4 constant is the string obtained by converting each hexadecimal digit to a 4-digit binary number as shown in the table below. A bit constant has the attribute BIT(n), where n is the length of the bit value represented by the constant.

| Hexadecimal Digit (B4) | Binary Digits (B or B1) |
|------------------------|-------------------------|
| 0 | 0000 |
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0011 |
| 4 | 0100 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |
| 8 | 1000 |
| 9 | 1001 |
| A | 1010 |
| B | 1011 |
| C | 1100 |
| D | 1101 |
| E | 1110 |
| F | 1111 |

Figure 12-2. Hexadecimal Digits, and the Binary Equivalents

The maximum length of a bit constant (including apostrophes and the B, B1, or B4 characters) is 512 characters.

A null bit constant is two consecutive apostrophes, followed immediately by B, B1, or B4. Its value is the null bit string.

Examples of bit constants and the lengths of the represented bit values are:

| Constant | Length |
|-------------------|--------|
| '1'B | (1) |
| '11111010110001'B | (14) |
| ''B | (0) |
| '10100011'B | (8) |
| '10100011'B1 | (8) |
| 'A3'B4 | (8) |

In the last three examples, the values are the same.

You declare a bit variable with the BIT attribute and a length specification. For example:

```
DECLARE SYMPTOMS BIT (64) ALIGNED;
```

In this statement, SYMPTOMS represents bit values 64 bits long.

You must declare all bit variables with the ALIGNED attribute. If you do not specify ALIGNED, the compiler issues a warning message and assumes ALIGNED.

Character Data

A **character value** is a sequence of characters. It can include any language character and extralingual character. Any blank included in a character value is included in the count of the length of the value. A character value is stored in consecutive bytes, each character occupying 1 byte.

A **character constant** is a sequence of characters enclosed in apostrophes. The value of a character constant is the string of characters between the enclosing apostrophes, except that if you want to represent an apostrophe within the string, you must write it in the constant as two consecutive apostrophes with no intervening blanks. A character constant has the attribute CHARACTER(n), where n is the length of the character value represented by the constant.

The maximum length of a character constant, including the enclosing apostrophes, and counting each apostrophe as a separate character, is 512.

Examples of character constants and the lengths of their values are:

| Constant | Length |
|----------------------------------|--------|
| 'LOGARITHM TABLE' | (15) |
| 'PAGE 5' | (6) |
| 'SHAKESPEARE''S ''''HAMLET'''''' | (24) |
| 'SHAKESPEARE''S ''HAMLET'''' | (22) |
| 'AC438-19' | (8) |
| '' | (0) |

The last example is the null character constant, which is written as two consecutive apostrophes and represents the null character string.

You declare a character variable with the CHARACTER attribute and a length specification. For example:

```
DECLARE USER CHARACTER (15);
```

In this statement, USER represents character values 15 characters long.

VARYING Attribute

The VARYING attribute specifies that the variable is to represent varying-length strings. When you specify VARYING, the length of the CHARACTER attribute specifies the maximum length.

The length at any time is the length of the current value. The storage allocated for varying-length strings is 2 bytes longer than the declared maximum length. The left-most 2 bytes hold the character string's current length in bytes.

The following DECLARE statement specifies that the name USER is to represent varying-length character data items with a maximum length of 15:

```
DECLARE USER CHARACTER (15) VARYING;
```

The length for USER at any time is the length of the data item assigned to it at that time. You can determine the length at any given time with the LENGTH built-in function.

PICTURE Data Attribute

Data declared with the PICTURE attribute is known as **picture data**. A picture data item has a numeric value, which is represented as a character value by means of the editing characters in the picture specification.

Only data that is, or can be converted to, an arithmetic value can be assigned to a picture variable.

```

▶▶PICTURE—'picture_specification'—▶▶
Abbreviation: PIC for PICTURE
    
```

picture_specification

A sequence of picture characters. It specifies the character form and arithmetic characteristics of the value of the picture variable.

There must be at least one blank between the keyword PICTURE and the first apostrophe.

The syntax of the picture specification is as follows:

PROBLEM DATA ATTRIBUTES

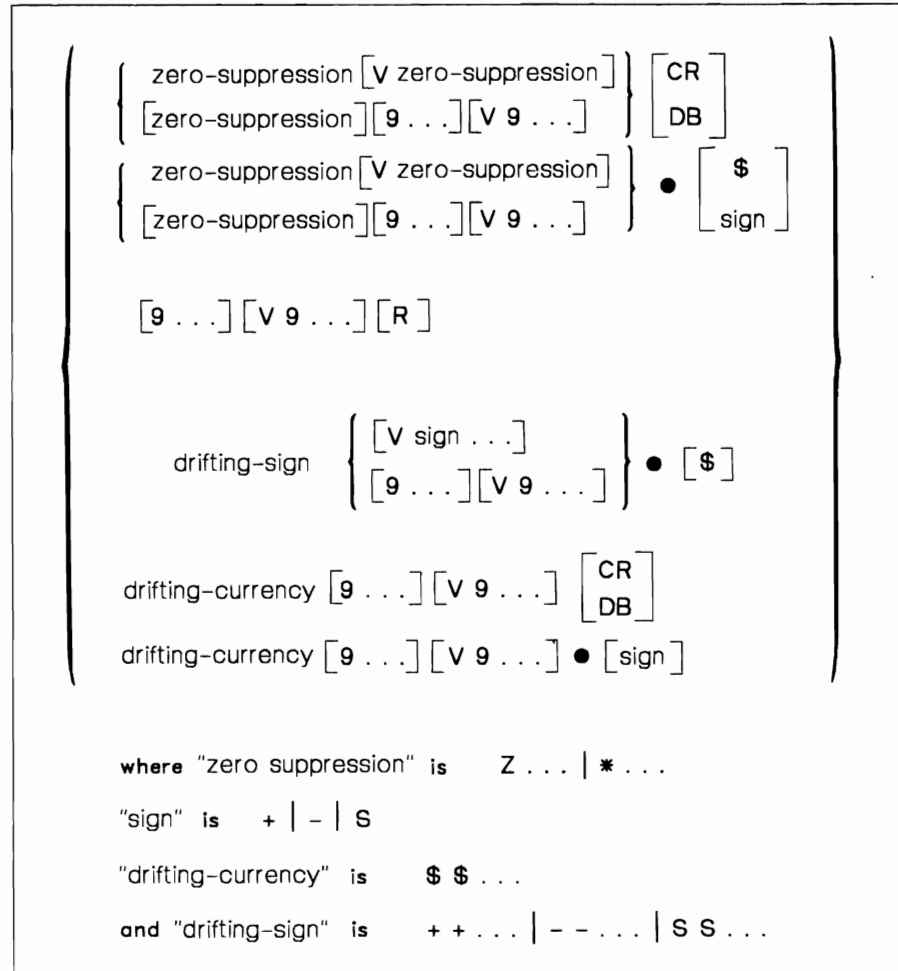


Figure 12-3. Picture Specification Syntax

The following should be noted:

- Only one type of sign character may be used.
- Only one type of zero suppression character may be used.
- The picture specification must contain at least one digit position.
- Insertion characters (, . / B), which are not shown in the syntax, can appear anywhere in a picture specification except within the character pairs CR (credit) or DB (debit).

Picture specifications may contain both uppercase and lowercase characters. The maximum length of a picture specification is 255 characters, which includes up to 15 digit positions, plus an optional V picture character.

Picture data items having only the characters 9, V, and R in the picture specification are represented in storage as zoned decimal numbers. Zoned decimal data is stored with the rightmost four bits of each byte holding a decimal digit in binary form. If the character R is the rightmost character in the picture specification, the leftmost four bits of the rightmost byte hold the sign indication. Arithmetical operations using picture data items of this type process more quickly than arithmetical oper-

ations using picture data items which include edit characters in the picture specification, because less conversion is required before the operation can be processed.

Like a coded arithmetic data item, a picture data item has a base, a scale, and a precision. The base of a picture data item is decimal, and its scale is fixed-point. The precision is derived from the picture specification. It is (p,q), where p, the number of digits, is the number of digit positions (conditional or unconditional) in the picture specification, and q, the scale factor, is the number of digit positions to the right of the V picture character; q = 0 if the V picture character is omitted.

The picture specification in the PICTURE attribute describes:

- The range of numeric values, expressed by the digit positions of the picture specification, the assumed decimal point, and possibly a sign.
- The representation as a character value, expressed by all the picture characters except the V, which specifies the assumed location of a decimal point. The character value is obtained by representing, or editing, the numeric value by means of the picture specification: the decimal digits in the character value are taken from the numeric value, and the editing characters are inserted as prescribed by the picture specification.

For example:

```
DECLARE PRICE1 PICTURE '999V99',
        PRICE2 PICTURE '$999V.99';
```

PRICE1 can represent numeric values in the range of 000.00 through 999.99; it cannot represent negative values. Any value assigned to PRICE1 is maintained as a zoned decimal value of 5 decimal digits, with an assumed decimal point preceding the rightmost two digits. A value assigned to PRICE1 is aligned on the assumed decimal point in the same way that decimal point alignment is maintained for fixed-point decimal data.

PRICE2 can represent the same range of numeric values. In the picture specification for PRICE2, the currency symbol (\$) and the decimal point (.) are editing characters. They are stored as characters in the data item. They are not, however, a part of its numeric value.

Picture data is stored in its character form. The use of its numeric value involves a conversion to decimal fixed-point. Depending on the context of a picture data item, its character value or its numeric value is used.

The character value is used when:

- The picture variable appears in an assignment to a character variable.
- A reference is made to a character variable that is based on a pointer to the picture variable.
- The value of a picture variable is transmitted by means of edit-directed output with the A format item or by record data transmission.

The numeric value is used when:

PROBLEM DATA ATTRIBUTES

- Arithmetic operations are processed on the picture variable.
- The picture variable is assigned to an arithmetic variable.
- The picture variable is used with the B, B1, B4, E, or F format items.

A value that is assigned to a picture variable will, if necessary, be truncated or extended with zeros or other characters (as prescribed by the picture specification) at the affected end. If significant digits or a negative sign are truncated, the result is undefined.

For example:

```
DECLARE PICPRICE PICTURE '$99V.99'.,  
        CHARPRICE CHARACTER (6),  
        DECPRICE FIXED DECIMAL (6,2);  
PICPRICE = 12.28;  
CHARPRICE = '$12.28';
```

In this example, after processing the second assignment statement, the character values of PICPRICE and CHARPRICE are identical. They would be printed in exactly the same way if they were printed in stream output by means of the A format item. PICPRICE and CHARPRICE produce different results, however, if they are used in an arithmetic context. Consider the following assignment statements:

```
DECPRICE = PICPRICE;
```

The value of DECPRICE is now 0012.28. The numeric value only is assigned: the currency symbol and decimal point are ignored, because they are editing symbols and not part of the arithmetic value.

```
CHARPRICE = PICPRICE;
```

The value of CHARPRICE is now '\$12.28'. Because the assignment is to a character variable, the editing characters are part of the the value assigned.

```
DECPRICE = CHARPRICE;
```

This assignment statement would cause an error. The value of CHARPRICE is not a valid arithmetic constant, because it includes a currency symbol.

```
PICPRICE = CHARPRICE;
```

This assignment statement would also cause an error. CHARPRICE includes a currency symbol, which makes it an invalid arithmetic constant.

Digit and Decimal Point Characters

The picture character 9 specifies a digit position; the picture character V specifies the assumed decimal point.

9 Specifies that the associated position in the data item is to contain a digit.

A string of 9s (picture characters) specifies that the item is to be represented by a character value of the same length as the string of 9s, each character of which is a digit (0 through 9). The numeric value is the value of the digits as an unsigned decimal number. For example:

```
DECLARE NUMBER PICTURE '999';  
NUMBER = 123;
```


In this example, the precision derived from the picture specification is (3,0). The character value of NUMBER is '123', and its numeric value is 123.

- V** Specifies that a decimal point is assumed at this position in the associated data item. But it does not specify to insert an actual decimal point. The integer and fractional parts of the assigned value are aligned on the V character; therefore, an assigned value may be truncated or extended with zero digits at either end. If no V character appears in the picture specification, a V is assumed at its right-hand end. This will truncate the assigned value to an integer. For example:

```
DECLARE VALUE PICTURE '99V999',
           CVALUE CHARACTER(5);
VALUE = 12.345;
CVALUE = VALUE;
```

In this example, the derived precision of VALUE is (5,3), its character value is '12345', and its numeric value is 12.345. On assignment to CVALUE, the character value is assigned. The resulting value of CVALUE is '12345'.

The statements

```
VALUE = 1.2;
CVALUE = VALUE;
```

result in the character value of VALUE being '01200' and its numeric value being 1.2. The resulting value of CVALUE is '01200'. To assign -0.1 or 123.4 to VALUE would, however, give an undefined result.

Figure 12-4 on page 12-27 gives examples of digit and decimal point picture specifications.

Zero Suppression Characters

The zero suppression picture characters specify conditional digit positions in the character value and replaces leading zeros with asterisks or blanks.

- Z** Replaces a leading zero in the associated data position with a blank character.
- *** Is used in the same way as the picture character Z, except that leading zeros are replaced by asterisks.

Figure 12-5 on page 12-27 gives examples of the use of zero suppression characters.

Insertion Characters

The insertion picture characters (, . / B) inserts the specified character (comma, period, slash, or blank) into the associated character value position of the picture data, if there is no zero suppression. They do not indicate digit positions, but are inserted between digits. Insertion characters are applicable only to the character value. They specify nothing about the numeric value of the data item.

If zero suppression occurs, the character is inserted only in the following cases:

- When an unsuppressed digit appears to the left of the insertion character

PROBLEM DATA ATTRIBUTES

- When a V appears immediately to the left of the insertion character and the fractional part contains a significant digit.
- When the insertion character is at the start of the picture specification or is preceded only by characters that do not specify digit positions.

In all other cases where zero suppression occurs, an insertion character is treated as though it were a zero suppression character.

The following example shows decimal conventions that are used in various countries:

```
DECLARE A PICTURE 'Z,ZZZ,ZZZV.99',  
        B PICTURE '*,***,***V,99',  
        C PICTURE '*B***B***V,99';
```

The derived precision of the picture variables A, B, and C is (9,2).

If A, B, and C are assigned the integer constant 1234, the character value of A, B, and C, respectively, will be

```
' 1,234.00'  
'***1.234,00'  
'***1 234,00'
```

Their numeric value is 1234.

The following example shows that the decimal point is aligned on the character V and not on the period insertion character:

```
DECLARE RATE PICTURE '9V99.99';  
RATE = 7.62;
```

The derived precision of RATE is (5,4), its character value is '762.00', and its numeric value is 7.62.

Figure 12-6 on page 12-28 gives examples of the use of insertion characters.

Sign and Currency Characters

The sign and currency picture characters are \$, S, +, and -. A fuller description of these characters is given below.

You can use these picture characters as either static or drifting characters.

The static use specifies that a sign, a currency symbol, or a blank appears in the associated position. A static character is specified by its single occurrence in the picture specification.

The drifting use specifies to suppress leading zeroes. A drifting character is specified by multiple use of that character in a picture specification. The position associated with the leftmost drifting character can contain only the drifting character or blank, never a digit. Therefore, if the drifting string contains the drifting character n times, the string is associated with $n-1$ conditional digit positions. After a data assignment to a picture variable, the rightmost suppressed position associated with the picture

character will contain a sign, a blank, or a currency symbol (except that where all digit positions are occupied by drifting characters and the value of the data item is zero, the drifting character is not inserted). For example:

```
DECLARE A PICTURE 'SSSV.9';
A = -2.0;
```

In this example, the derived precision of A is (3,1), its character value is ' -2.0', and its numeric value is -2.0.

Any insertion character within or immediately following the string is considered part of the drifting string. A V ends the drifting string, except when the numeric value of the data item is zero; in that case, the character value will be all blanks (except for any insertion characters to the left of the drifting string).

In the data item, the character value position associated with an insertion character in a string of drifting characters contains one of the following:

- The insertion character, if there is a significant digit to the left
- The drifting character, if the next position to the right contains the leftmost significant digit of the value
- Blank, if the leftmost significant digit of the value is more than one position to the right.

The sign and currency characters are as follows:

\$ Specifies the currency symbol. For example:

```
DECLARE PRICE PICTURE '$99V.99';
PRICE=12.45;
```

The derived precision of PRICE is (4,2), its character value is '\$12.45', and its numeric value is 12.45.

S Specifies the plus sign character (+) if the numeric value is greater than or equal to zero; otherwise it specifies the minus sign character (-). For example:

```
DECLARE ROOT PICTURE 'S999';
```

50 is held as '+050', zero as '+000', and -50 as '-050'.

- +** Specifies the plus sign character (+) if the numeric value is greater than or equal to zero; otherwise it specifies a blank.
- Specifies the minus sign character (-) if the numeric value is less than zero; otherwise it specifies a blank.

If, in an assignment to a picture variable, the fractional digits are truncated so that the resulting numeric value is zero, the sign inserted in the data item corresponds to the value of the data item prior to its truncation. Therefore, the sign in the picture data may depend on how the value was calculated.

Figure 12-7 on page 12-29 gives examples of the use of drifting picture characters.

PROBLEM DATA ATTRIBUTES

Credit and Debit Characters

The character pairs CR (credit) and DB (debit) specify the signs of picture data.

- CR** Specifies that the associated positions will contain the letters CR if the numeric value is less than zero; otherwise, the positions will contain two blanks.
- DB** Is used in the same way as CR, except that the letters DB appear in the associated positions.

Figure 12-8 on page 12-30 gives examples of the CR and DB picture characters.

Digit and Signed Character

The digit and signed character R specifies that the associated position will contain an EBCDIC character or a digit, depending on the sign of the data item.

- If the arithmetic value of the data item is less than zero, the associated position contains an EBCDIC character. This EBCDIC character represents the negative value of the corresponding digit shown in the table below.
- If the arithmetic value of the data item is greater than or equal to zero, the associated position contains a digit.

The associated characters and digits are shown in the following table:

| EBCDIC character | Digit |
|------------------|-------|
| } | 0 |
| J | 1 |
| K | 2 |
| L | 3 |
| M | 4 |
| N | 5 |
| O | 6 |
| P | 7 |
| Q | 8 |
| R | 9 |

For example:

```
DECLARE INTEGER PICTURE '99R';  
READ FILE (INFILE) INTO (INTEGER);
```

will set INTEGER to 321 if '321' is found in the next record and will set INTEGER to -321 if '32J' is found.

| Source Data | Picture Specification | Derived Precision | Character Value | Numeric Value |
|-------------|-----------------------|-------------------|-----------------|---------------|
| 12345 | 99999 | (5,0) | 12345 | 12345 |
| 123 | 99999 | (5,0) | 00123 | 123 |
| 123.45 | 999V99 | (5,2) | 12345 | 123.45 |
| 123.45 | 99999 | (5,0) | 00123 | 123 |

Figure 12-4. Digit and Decimal Point Examples

| Source Data | Picture Specification | Derived Precision | Character Value | Numeric Value |
|-------------|-----------------------|-------------------|-----------------|---------------|
| 12345 | <i>ZZZ99</i> | (5,0) | 12345 | 12345 |
| 100 | <i>ZZZ99</i> | (5,0) | bb100 | 100 |
| 100 | <i>ZZZZZ</i> | (5,0) | bb100 | 100 |
| 0 | <i>ZZZZZ</i> | (5,0) | bbbbb | 0 |
| 123.45 | <i>ZZZ99</i> | (5,0) | bb123 | 123 |
| 1.23 | <i>ZZZV99</i> | (5,2) | bb123 | 1.23 |
| 0.08 | <i>ZZZVZZ</i> | (5,2) | bbb08 | 0.08 |
| 0 | <i>ZZZVZZ</i> | (5,2) | bbbbb | 0 |
| 100 | <i>*****</i> | (5,0) | **100 | 100 |
| 0 | <i>*****</i> | (5,0) | ***** | 0 |
| 0.01 | <i>***V**</i> | (5,2) | ***01 | 0.01 |
| 95 | <i>\$\$\$9.99</i> | (5,0) | \$\$\$0.95 | 95 |
| 12350 | <i>\$\$\$9.99</i> | (5,0) | \$123.50 | 12350 |

Figure 12-5. Examples of Zero Suppression

Note: In this figure, the letter b indicates a blank character.

PROBLEM DATA ATTRIBUTES

| Source Data | Picture Specification | Derived Precision | Character Value | Numeric Value |
|-------------|-----------------------|-------------------|-----------------|---------------|
| 1234 | 9,999 | (4,0) | 1,234 | 1234 |
| 1234.56 | 9,999V.99 | (6,2) | 1,234.56 | 1234.56 |
| 12.34 | ZZ.VZZ | (4,2) | 12.34 | 12.34 |
| 0.03 | ZZ.VZZ | (4,2) | bbb03 | 0.03 |
| 0.03 | ZZV.ZZ | (4,2) | bb.03 | 0.03 |
| 12.34 | ZZV.ZZ | (4,2) | 12.34 | 12.34 |
| 0 | ZZV.ZZ | (4,2) | bbbbbb | 0 |
| 1234567.89 | 9,999,999V.99 | (9,2) | 1,234,567.89 | 1234567.89 |
| 12345.67 | ** ,999V.99 | (7,2) | 12,345.67 | 12345.67 |
| 123.45 | ** ,999V.99 | (7,2) | ***123.45 | 123.45 |
| 1234567.89 | 9.999.999V,99 | (9,2) | 1.234.567,89 | 1234567.89 |
| 123456 | 99/99/99 | (6,0) | 12/34/56 | 123456 |
| 123456 | 99.9/99.9 | (6,0) | 12.3/45.6 | 123456 |
| 1234 | ZZ/ZZ/ZZ | (6,0) | bbb12/34 | 1234 |
| 12 | ZZ/ZZ/ZZ | (6,0) | bbbbbb12 | 12 |
| 0 | ZZ/ZZ/ZZ | (6,0) | bbbbbbbbb | 0 |
| 0 | **/**/** | (6,0) | ***** | 0 |
| 0 | **B**B** | (6,0) | ***** | 0 |
| 123456 | 99B99B99 | (6,0) | 12b34b56 | 123456 |
| 123 | 9BB9BB9 | (3,0) | 1bb2bb3 | 123 |
| 12 | 9BB/9BB | (2,0) | 1bb/2bb | 12 |

Figure 12-6. Examples of Insertion Characters

Note: In this figure, the letter b indicates a blank character.

| Source Data | Picture Specification | Derived Precision | Character Value | Numeric Value |
|-------------|-----------------------|-------------------|-----------------|---------------|
| 123.45 | \$999V.99 | (5,2) | \$123.45 | 123.45 |
| 12 | 99\$ | (2,0) | 12\$ | 12 |
| 1.23 | \$ZZZV.99 | (5,2) | \$bb1.23 | 1.23 |
| 0 | \$ZZZV.ZZ | (5,2) | bbbbbbb | 0 |
| 0 | \$\$\$.\$\$ | (4,0) | bbbbbb | 0 |
| 123.45 | \$\$\$9V.99 | (5,2) | \$123.45 | 123.45 |
| 1.23 | \$\$\$9V.99 | (5,2) | bb\$1.23 | 1.23 |
| 12 | \$\$\$,999 | (5,0) | bbb\$012 | 12 |
| 1234 | \$\$\$,999 | (5,0) | b\$1,234 | 1234 |
| 2.45 | SZZZV.99 | (5,2) | +bb2.45 | 2.45 |
| 214 | SS,SS9 | (4,0) | bb+214 | 214 |
| -4 | SS,SS9 | (4,0) | bbbb-4 | -4 |
| -123.45 | -999V.99 | (5,2) | -123.45 | -123.45 |
| 123.45 | 999V.99S | (5,2) | 123.45+ | 123.45 |
| 1.23 | ++B+9V.99 | (5,2) | bbb+1.23 | 1.23 |
| 1.23 | ---9V.99 | (5,2) | bbb1.23 | 1.23 |
| -1.23 | SSS9V.99 | (5,2) | bb-1.23 | -1.23 |

Figure 12-7. Examples of Signs and Currency Symbols

Note: In this figure, the letter b indicates a blank character.

PROGRAM CONTROL DATA ATTRIBUTES

| Source Data | Picture Specification | Derived Precision | Character Value | Numeric Value |
|-------------|-----------------------|-------------------|-----------------|---------------|
| -123 | \$Z.99CR | (3,0) | \$1.23CR | -123 |
| 12.34 | \$ZZV.99CR | (4,2) | \$12.34bb | 12.34 |
| -12.34 | \$ZZV.99DB | (4,2) | \$12.34DB | -12.34 |
| 12.34 | \$ZZV.99DB | (4,2) | \$12.34bb | 12.34 |

Figure 12-8. Examples of CR and DB Picture Characters

Note: In this figure, the letter b indicates a blank character.

Program Control Data Attributes

Program control data attributes refer to pointer, label, entry, and file data. Scope and storage attributes can be specified for program control data.

POINTER Attribute

| |
|--|
| ▶—POINTER—◀ Abbreviation: PTR for POINTER |
|--|

A **pointer value** identifies the location of data in storage. You declare a pointer variable with the POINTER attribute. You can also specify VARIABLE, scope or storage attributes, with the default values shown in Figure 12-1 on page 12-3. Pointer variables can be grouped into aggregates.

A pointer value may be obtained by one of the following means:

- The ADDR or NULL built-in function.
- A READ statement with the SET option.
- The ALLOCATE statement.

A pointer value can also be obtained by means of a parameter from some built-in subroutines.

Pointer values can be assigned, compared ($=$ or \neq), passed as arguments, or returned by a function. They cannot be converted or specified in operations.

If you use a pointer as the target of an assignment statement, the source must be either another valid pointer, or the ADDR or NULL built-in function.

You can use a pointer, together with a based variable, to access the location in storage identified by the pointer value (see "Based Variable Reference and Pointer Qualification" on page 5-20).

Pointer Built-In Functions

The ADDR built-in function, when applied to a variable, returns a pointer value that identifies the location of the variable in storage.

The NULL built-in function returns a null pointer value, which does not identify the location of any variable. You use this value when a pointer variable should not identify a location in storage. A pointer variable acquires the null pointer value by assignment of the value of the NULL built-in function.

LABEL Attribute

A **label** identifies a statement in the running program. There are two kinds of labels: label constants and label variables.

A **label constant** is a name written as the label prefix of any statement other than PROCEDURE. During processing, program control can be transferred to the statement by referring to its label prefix.

A name is explicitly declared as a label constant by its appearance as a label.

In the example:

```
ABCDE: MILES = SPEED * HOURS;
```

ABCDE is a label constant. The statement can be processed either by normal sequential processing of instructions or by transferring control to it from some other point in the program by means of a GO TO statement.

You declare a **label variable** by specifying the LABEL attribute.



When you use the LABEL attribute, you can also specify VARIABLE, scope, and storage attributes (although INITIAL is not valid). Defaults are shown in Figure 12-1 on page 12-3. Label values can also be grouped into aggregates. You set a label variable by assignment.

You can use a label variable with the GO TO statement. Control is transferred to the statement identified by the value of the label variable. For example:

```
DECLARE LABEL1 LABEL VARIABLE;
  STMT1: ITEM1 = ITEM2;
      .
      .
  STMT2: ITEM1 = ITEM3;
      .
      .
LABEL1 = STMT1;
GO TO LABEL1;
```

PROGRAM CONTROL DATA ATTRIBUTES

STMT1 and STMT2 are label constants, and LABEL1 is a label variable. After STMT1 has been assigned to LABEL1, the statement GO TO LABEL1 transfers control to the statement labeled STMT1. Elsewhere, the program could contain a statement LABEL1 = STMT2. Any reference to LABEL1 would then be the same as a reference to STMT2. This value of LABEL1 is retained until another value is assigned; but it becomes invalid if the block containing the statement labeled STMT2 becomes inactive.

ENTRY Attribute

You use the ENTRY attribute to declare an entry variable or an external entry constant and to describe the attributes of any parameters the associated entry value may have.

An **entry data item** represents a procedure. You refer to an entry data item by means of an entry reference.

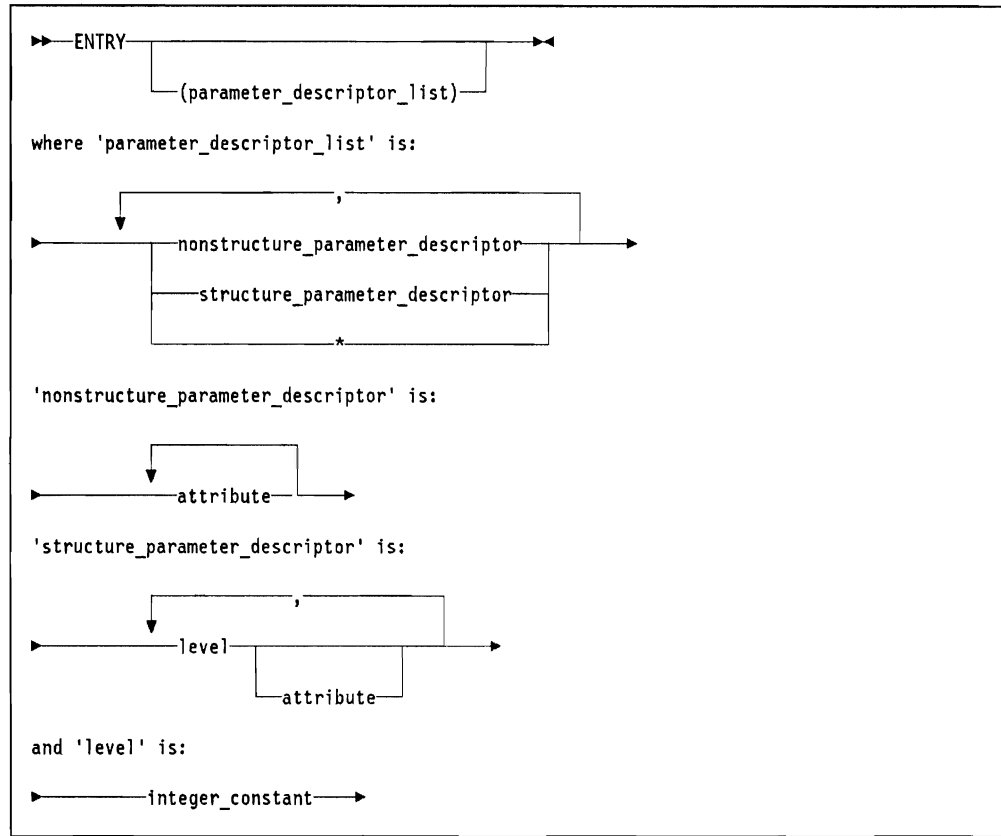
An **entry reference** is an entry constant, an entry variable reference, or a function reference that returns an entry value.

An **entry constant** is the label prefix to a PROCEDURE statement. If the procedure is external, you must declare the entry constant with the ENTRY attribute. If the procedure is a function, you must declare the entry constant with the RETURNS attribute.

An **entry variable** is a variable to which an entry value can be assigned. It is declared with the ENTRY and VARIABLE attributes and, for a function, with the RETURNS attribute as well. You cannot declare an internal entry constant in a DECLARE statement. You can also specify scope and storage attributes (except INITIAL). Defaults for these are shown in Figure 12-1 on page 12-3. Entry variables can be grouped into aggregates.

When an entry constant which is an entry point of an internal procedure is assigned to an entry variable, the assigned value remains valid only for as long as the block that the entry constant was internal to remains active.

The syntax of the ENTRY attribute is as follows:



parameter_descriptor_list

A list of parameter_descriptors, each of which gives a set of attributes for the parameter it corresponds to positionally in the procedure.

Parameter_descriptors are separated by commas.

The number of parameter_descriptors in the ENTRY attribute and the attributes specified for each parameter must match the parameters as declared in the procedure represented by the entry name. An ENTRY attribute without a parameter_descriptor_list describes a procedure with no parameters.

nonstructure_parameter_descriptor

For a nonstructure_parameter_descriptor, you must specify at least one data attribute other than FILE. This corresponds to the rules for giving attributes to a parameter in a DECLARE statement (see "Parameter Attributes" on page 14-3). The attributes are separated by blanks. If you specify the ENTRY attribute as a parameter_descriptor, it must not have a parameter_descriptor or a RETURNS attribute.

structure_parameter_descriptor

Each level number, together with any attributes specified for it, is separated from the next by a comma. You can specify the same attributes for a field name as you can for a nonstructure_parameter_descriptor.

The level numbers need not be the same as those of their corresponding parameters, but they must be in the same order with identical structuring.

PROGRAM CONTROL DATA ATTRIBUTES

IBM Extension

*

This `parameter_descriptor` is valid only when the entry variable or entry constant represents a non-PL/I routine, that is, one for which you have coded `OPTIONS (ASSEMBLER)`. It specifies that no check is made against this parameter of the associated non-PL/I routine.

End of IBM Extension

The rules for specifying string lengths or array bounds in a `parameter_descriptor` are the same as for parameter lengths and bounds in a `DECLARE` statement.

For arrays, the `parameter_descriptor` must include the array dimensions in parentheses as the first attribute. For example, if the PL/I program `PROC1` had parameters coded as follows:

```
PROC1: PROCEDURE (EMPLOYEE, COMMISSIONS, NAME);
        DECLARE EMPLOYEE          FIXED DECIMAL (5),
               COMMISSIONS(20)    FIXED DECIMAL (7),
               NAME                CHARACTER (*);
```

the `ENTRY` declaration would be as follows:

```
DECLARE PROC1 ENTRY (FIXED DECIMAL (5),
                   (20) FIXED DECIMAL (7),
                   CHARACTER (*));
```

The following example illustrates the use of `structure_parameter_descriptors` in an external procedure:

```
TEST: PROCEDURE
      (AFID, BFIB, CSTRUC, DSTRUC, ECHAR, P);
      DECLARE AFID FIXED DECIMAL(5),
             BFIB FLOAT BINARY (15),
             1 CSTRUC,
               5 QCHAR CHARACTER (3),
               5 RSTRUC,
                 10 SFID FIXED DECIMAL (5),
             1 DSTRUC,
               5 XCHAR CHARACTER (3),
               5 YSTRUC,
                 10 ZFID FIXED DECIMAL (5),
             ECHAR(*) CHARACTER (10),
             P POINTER;
```

```
      . . .
      END TEST;
```

To call this procedure, these `structure_parameter_descriptors` could be declared as follows:

PROGRAM CONTROL DATA ATTRIBUTES

```
DECLARE TEST ENTRY
    (DECIMAL FIXED (5),
     BINARY FLOAT (15),
     1,
     5 CHARACTER (3),
     5,
     10 DECIMAL FIXED (5),
     1,
     5 CHARACTER (3),
     5,
     10 FIXED DECIMAL (5),
     (*) CHARACTER (10),
     POINTER);
```

The use of the parameter_descriptor * is illustrated in the following example:

```
DECLARE TRANSFER ENTRY
    (CHARACTER(10),*,*,FIXED DECIMAL(5))
    OPTIONS(ASSEMBLER);
```

The declaration indicates four parameters of a non-PL/I routine, with attributes specified for the first and fourth parameter.

No parameter_descriptor_list can be defined for an ENTRY attribute specified in a parameter_descriptor_list for an ENTRY variable. For example, the following declaration is not valid:

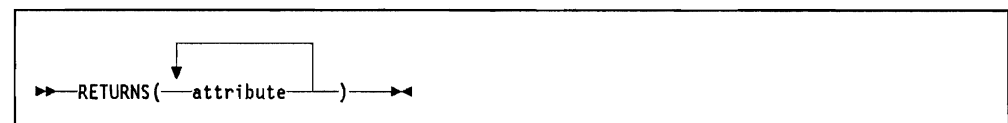
```
DECLARE SUBPROC ENTRY (FIXED BINARY (15),
                      FLOAT DECIMAL (7),
                      ENTRY (CHARACTER (10)) );
```

The parameter_descriptor_list for the ENTRY attribute would have to be specified in SUBPROC:

```
SUBPROC: PROCEDURE (BINARYITEM, DECIMALITEM,ENTRYITEM);
    .
    .
    .
    DECLARE ENTRYITEM ENTRY (CHARACTER(10));
```

RETURNS Attribute

You specify the RETURNS attribute in a DECLARE statement for an entry variable or external entry name that represents a function. It specifies the attributes of the returned value.



attribute

The valid attributes are the same as those for the RETURNS option (see "PROCEDURE Statement" on page 14-2).

BUILTIN ATTRIBUTE

The attributes specified in the RETURNS attribute for an entry variable or an external entry constant must agree with those specified in the RETURNS option of the corresponding PROCEDURE statement.

OPTIONS(ASSEMBLER) Attribute

The OPTIONS(ASSEMBLER) attribute allows you to communicate with a non-PL/I program.

▶▶—OPTIONS(ASSEMBLER)—◀◀

Abbreviation: ASM for ASSEMBLER

ASSEMBLER

Specifies that the designated entry point is a program written in an AS/400 language other than PL/I. PL/I will pass arguments directly to the program, rather than through PL/I control blocks. Entries with the ASSEMBLER option are subject to the following rules:

- They cannot be used as a function reference.
- Any number of arguments can be passed in the CALL statement calling the entry, from zero up to the number specified by the entry declaration, but intervening arguments cannot be omitted.

BUILTIN Attribute

The BUILTIN attribute specifies that the declared name denotes a built-in function, a pseudovisible, or a built-in subroutine.

▶▶—BUILTIN—◀◀

You only need to use the BUILTIN attribute when you are using the name of a built-in function or subroutine (see Chapter 15, “Built-In Functions, Subroutines, and Pseudovisibles”) as a user-defined name in a different block.

When you use a built-in name as a user-defined name, you declare it again with the BUILTIN attribute in any other block to associate it again with the built-in function or built-in subroutine. Consider the following examples:

Example 1:

```

A: PROCEDURE;
  DECLARE (SQRT(20),P) FLOAT BINARY (20);
  . . .
  X = SQRT(P);
  . . .
B: BEGIN;
  DECLARE SQRT BUILTIN;
  Z = SQRT(P);
  END B;
  . . .
END A;

```

Example 2:

```

A: PROCEDURE;
  DECLARE P FIXED DECIMAL (7,2);
  SQRT: PROC(PARAM) RETURNS (FIXED(7,2));
  DECLARE PARAM FIXED (13);
  . . .
  END SQRT;
  . . .
  X = SQRT(Y);
  . . .
B: BEGIN;
  DECLARE SQRT BUILTIN;
  Z = SQRT(P);
  END B;
  . . .
END A;

```

In A of both examples, SQRT is a user-defined name. In the assignment to the variable X, SQRT is a reference to the user-defined name SQRT. In B of both examples, SQRT is declared with the BUILTIN attribute. Any reference in B to SQRT is recognized as a reference to the built-in function and not to the user-defined name SQRT declared in A. For information on using built-in functions, subroutines, and pseudovariables, see Chapter 15, "Built-In Functions, Subroutines, and Pseudovariables."

VARIABLE Attribute

You can use the VARIABLE attribute to declare a variable of any type except FILE.

►—VARIABLE—◄

VARIABLE is implied for parameters, structures, and structure members, by any scope, storage class, or alignment attribute, and by some of the data type attributes (the default rules are given in "Names" on page 4-12). Constant is implied for label prefixes, and by ENTRY, unless VARIABLE is implied by other attributes. You

AGGREGATE DATA DECLARATIONS

must use the VARIABLE attribute to declare an entry variable if VARIABLE is not implied by any other attribute.

Aggregate Data Declarations

All data types, except file or entry constants, can be grouped into aggregates. The types of aggregates are arrays and structures. Single data items, called scalars, can be grouped into arrays or structures. Single variables, called scalar variables, can be grouped into array variables or structure variables.

An **array** is a collection, into one or more dimensions, of one or more array-elements with identical attributes. An **array-element** can be a scalar variable or a structure. Only the array itself is given a name. An individual item of an array is referred to by giving its subscript.

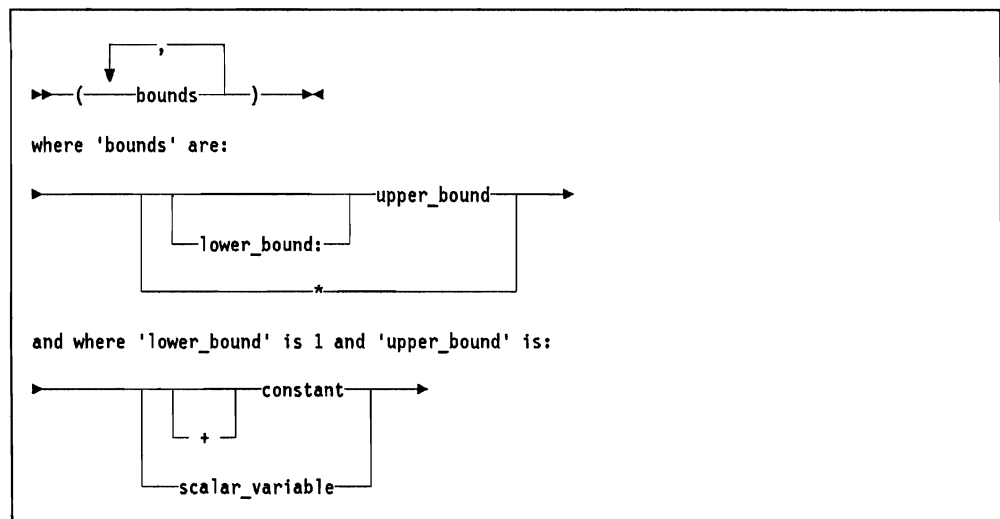
An array is declared with the dimension attribute.

A **structure** is a collection of data items that need not have identical attributes. Like an array, the entire structure is given a name, which can be used to refer to the entire aggregate of data. But, unlike an array, each field of a structure also has a name.

You use **level numbers** to specify the organization of a structure in a DECLARE statement.

Arrays and the Dimension Attribute

The **dimension attribute** specifies the number of dimensions of an array and indicates the bounds of each dimension. It must immediately follow the array name or factored list of array names.



bounds

The `upper_bound` or `upper` and `lower_bounds`. The number of bounds specifications indicates the number of dimensions in the array. For a parameter, you

can specify an asterisk to indicate that the lower and upper_bound are taken from the associated argument in the calling procedure.

lower_bound

The beginning of the dimension. If you specify the lower_bound, it must be an optionally signed integer constant whose value is 1. If you omit the lower_bound, 1 is assumed by default.

upper_bound

The end of the dimension. The upper_bound must be less than or equal to 32 767. You can specify the upper_bound as follows:

- If the variable is static, based, a member of a structure, or a parameter, the upper_bound must be an optionally signed integer constant.
- If the variable is automatic, the upper_bound must be an optionally signed integer constant or an unsubscripted non-based integer variable reference.
- The value of the upper_bound must be greater than or equal to 1.

The **extent** of a dimension is the number of integers between the lower and upper_bounds, including the bounds.

The maximum number of dimensions is 15. The total length of an array must not exceed 4194304 bytes (4 megabytes).

For a discussion and examples of how to use the dimension attribute to declare arrays, see “Using Arrays and the Dimension Attribute” on page 5-1.

Structures and Level Numbers

You can specify the organization of a structure in a DECLARE statement by preceding the names with **level numbers**. The major structure name must be declared with the level number 1, and minor structures and field names with level numbers greater than 1. Level numbers must be integer constants.

The level numbers you choose for successively deeper levels need not be the immediately succeeding integers. A minor structure at level n contains all the names with level numbers greater than n that lie between that minor structure name and the next name with a level number less than or equal to n .

The description of a major structure is ended by the declaration of another item with the level number 1, by the declaration of another item with no level number, or by the end of the DECLARE statement or descriptor list.

The maximum depth of logical levels is 15, and the highest valid level number is 255. The maximum length of a structure is 32 767 bytes.

For a discussion of how to use level numbers to describe structures, see “Using Arrays and the Dimension Attribute” on page 5-1.

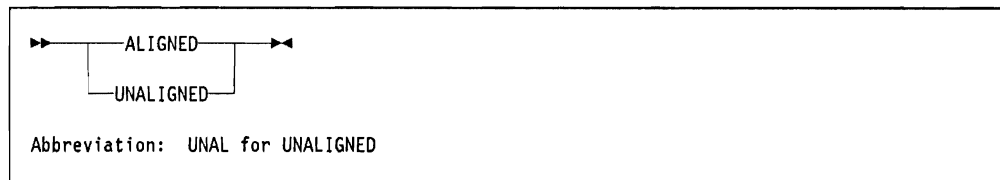
A structure name, either major or minor, can be given a dimension attribute in a DECLARE statement to declare an array of structures. An **array of structures** is an array whose elements are structures that have identical names, levels, and element

SCOPE ATTRIBUTES

attributes. For a discussion of how to use arrays of structures, see "Arrays of Structures" on page 5-5.

Alignment Attributes

By means of the **ALIGNED** and **UNALIGNED** attributes, you can choose to align data on the appropriate boundary. You can specify alignment attributes for scalars and arrays only.



ALIGNED

Specifies that the data item is aligned on the storage boundary corresponding to its data type requirement. For example, **BIN (15)** data is aligned on a halfword boundary and **BIN (31)** data on a fullword boundary. See "Data Mapping" on page 5-9 for a definition of these requirements.

Full Language Extension

UNALIGNED

Specifies that the data need not be aligned. Although the **UNALIGNED** attribute can reduce storage requirements, it may increase run time.

End of Full Language Extension

Bit data must be declared as **ALIGNED**.

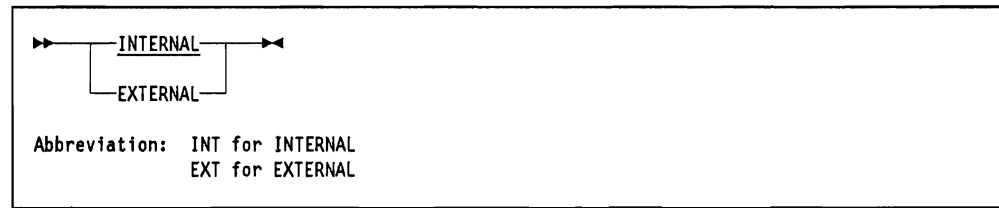
The default for character data and picture data is **UNALIGNED**. **UNALIGNED** can also be specified for fixed-point binary, floating-point binary, and floating-point decimal data.

For all other data types, the default is **ALIGNED**. **ALIGNED** can also be specified for character varying data.

For a discussion of how to use the alignment attributes, see "Data Alignment and the Alignment Attributes" on page 5-7.

Scope Attributes

You can use the **INTERNAL** and **EXTERNAL** attributes to specify the scope of a name.



INTERNAL

The default for variables with any storage class and for members of structures. Entry names of internal procedures are always internal.

EXTERNAL

The default for file and entry constants. Entry names of external procedures are always external.

For structures, INTERNAL and EXTERNAL should be specified in level 1 declarations only. An error message is sent if a member of a structure has a scope specified that is different from the scope specified in the level 1 declaration.

Storage Attributes

The declaration of a variable includes a storage class attribute either by explicit specification or by default.

The way storage is allocated for a variable, and the degree of control you can exercise over storage, are determined by the storage class of that variable. There are three storage classes: static, automatic, and based. Each is specified by its corresponding storage class attribute.

You can specify the storage class for level-one variables only. Elements of arrays and members of structures inherit the storage class of the array or structure.

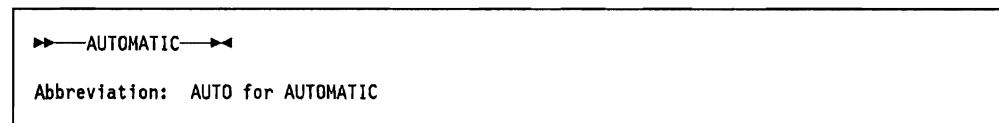
You cannot specify a storage class for a parameter or a named constant.

The default is AUTOMATIC for internal variables and STATIC for external variables.

Automatic and based variables can have internal scope only. Static variables can have either internal or external scope.

AUTOMATIC Attribute

You declare an automatic variable with the AUTOMATIC attribute.



Automatic variables can have internal scope only.

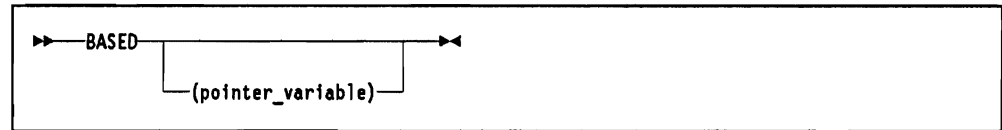
STORAGE ATTRIBUTES

AUTOMATIC is the default for internal variables.

For a discussion of the use of the AUTOMATIC attribute, see “Using the AUTOMATIC Attribute” on page 5-18.

BASED Attribute

You declare a based variable with the BASED attribute.



pointer-variable

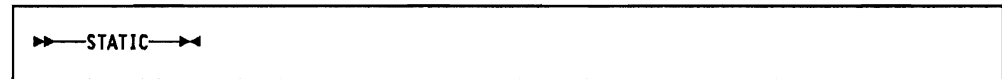
A simple non-based variable with the POINTER attribute.

Based variables can have internal scope only.

For a discussion of the use of the BASED attribute, see “Using the BASED Attribute” on page 5-19.

STATIC Attribute

You declare a static variable with the STATIC attribute.



Any expressions that specify lengths or bounds for a static variable must be integer constants.

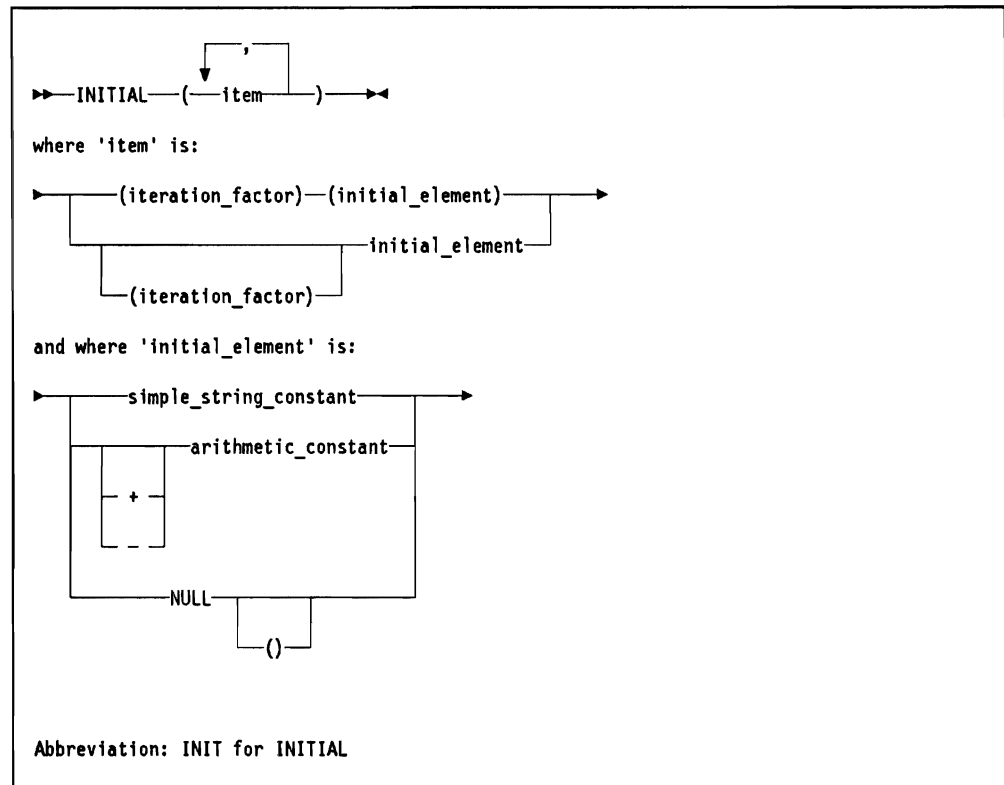
Static variables can have either internal or external scope.

STATIC is the default for external variables.

For a discussion of the STATIC attribute, see “Using the STATIC Attribute” on page 5-16.

INITIAL Attribute

The INITIAL attribute, used with the STATIC attribute, specifies values assigned to a scalar or array variable when storage is allocated to it.



iteration_factor

An integer constant in the range 1 through 32 767.

simple_string_constant

Must be in parentheses when preceded by an iteration factor.

Any variable declared with the INITIAL attribute may only be of type arithmetic, string, PICTURE, or POINTER.

You can specify the INITIAL attribute for arrays that do not have inherited dimensions, as well as for scalar variables. In a structure declaration, you can specify the INITIAL attribute only for field names.

You can specify only one initial value for a scalar variable, but more than one for an array variable. A structure variable requires separate initialization of each of its field names if they are scalar or array variables.

For a discussion of the use of the INITIAL attribute, see "Using the INITIAL Attribute" on page 5-17.

STORAGE ATTRIBUTES



Chapter 13. General PL/I Statements

This chapter describes the statements listed below:

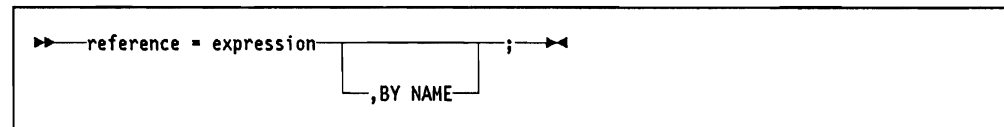
Assignment
 DO
 END
 GO TO
 IF
 ITERATE
 LEAVE
 Null
 OTHERWISE
 SELECT
 STOP
 WHEN

Assignment Statement

The assignment statement evaluates an expression and assigns its result to a target variable, which can be a scalar, array, or structure variable or a pseudovisible.

You can assign problem data of any type to a problem data variable or pseudovisible. All valid conversions are described in "Data Conversion" on page 5-27.

You can assign program control data only to a program control variable of the same data type.



A reference is described in Figure 9-1 on page 9-2.

A **scalar assignment** is processed as follows:

1. Subscripts and pointer qualifications on the left-hand side are evaluated.
2. The expression on the right-hand side is evaluated.
3. The value of the expression is converted, if necessary, to the attributes of the variable on the left-hand side according to the rules for data conversion. The converted value is then assigned to the variable on the left-hand side (see "Data Assignment" on page 5-24).

An **aggregate assignment** is processed if the left-hand side is an array or structure variable. Only the following cases are valid:

ASSIGNMENT

- The left-hand side is an array that is not an array of structures and the right-hand side is a scalar expression.

The right-hand side value will be assigned to each array element.

- The right-hand side is an aggregate variable whose size, shape, and component data types are identical to those of the left-hand side variable.

All scalar data of the right-hand side variable will be assigned to the corresponding elements or fields of the left-hand side variable. Because the component data types are identical, no data conversions will occur.

- The left-hand side is a structure and the right-hand side is a scalar variable.

The scalar variable is assigned to each member of the structure, with conversions processed where necessary. If an array is a structure member, the scalar variable is assigned to each element of the array.

Examples of Assignment Statements

The examples in this section use the following declarations:

```
DECLARE INDEX1          BINARY;
DECLARE (ARRAY1,ARRAY2)(10) BINARY,
        ARRAY3(2,3,4)   BINARY;
DECLARE 1 STRUCTURE1(5),
        5 BINFIXED1     BINARY,
        5 BINFLOAT1    FLOAT,
        1 STRUCTURE2(5),
        5 BINFIXED1     BINARY,
        5 BINFLOAT2    FLOAT;
```

An example of a scalar assignment is:

```
ARRAY3(INDEX1,1,2) = ARRAY2(INDEX1) + INDEX1;
```

The right-hand side is an operational expression, and the left-hand side is a scalar variable.

The following are examples of aggregate assignments:

An example of a scalar to array assignment is:

```
ARRAY3 = 0;
```

An example of a structure to structure assignment is:

```
STRUCTURE1(INDEX1) = STRUCTURE2(INDEX1+1);
```

An example of an array to array assignment is:

```
ARRAY1 = ARRAY2;
```

An example of an array of structures to array of structures assignment is:

```
STRUCTURE1 = STRUCTURE2;
```


BY NAME ASSIGNMENT

In a BY NAME assignment, each target must be a structure or an array of structures. A member of a target structure derives its value from those structure members in the expression that have the same name. Only those members of a target for which a member with the same name exists in the other target are affected. For example:

```

DECLARE 1 ORDERLIST,
        5 ITEMNAME    CHARACTER (30),
        5 ITEMNUMBER  CHARACTER (5),
        5 SUPPLIER    CHARACTER (15),
        5 PRICE       FIXED DECIMAL (5,2),
1 INVENTORY,
        5 ITEMNAME    CHARACTER (30),
        5 ITEMNUMBER  CHARACTER (5),
        5 QUANTITY    FIXED DECIMAL (4),
        5 LOCATION    CHARACTER (15);
ORDERLIST = INVENTORY, BY NAME;

```

The only two members the two structures have in common are ITEMNAME and ITEMNUMBER. Therefore, the effect of the BY NAME assignment is:

```

ORDERLIST,ITEMNAME = INVENTORY,ITEMNAME;
ORDERLIST,ITEMNUMBER = INVENTORY,ITEMNUMBER;

```

Just as in an ordinary assignment, the expression on the right is fully evaluated before any part of it is assigned.

For a BY NAME assignment involving scalars or arrays, the scalar or array enters into all of the implied assignments. For a BY NAME assignment involving nested structures, the name selection rules are applied recursively.

The rules for BY NAME assignment can be stated more precisely. A BY NAME assignment is carried out in four steps:

1. A by-name-parts-list is generated. The by-name-parts-list enumerates those structure elements that participate in the assignment.
2. The by-name-parts-list is used to select from and reorder the members of both structures involved in the assignment.
3. The expression is evaluated, using just those parts selected. Those parts whose names correspond are combined.
4. The value of the expression is transferred to the target, with named parts of the value going to the corresponding named parts of the target.

The by-name-parts-list is generated by first considering both variables that appear in the assignment. For each variable that is a structure or an array of structures, list the fully-qualified names of the structure members at every level, omitting the variable name from each fully-qualified name. (Scalars or arrays of scalars do not count.) The by-name-parts-list consists of just those names that are common to both of the lists from the individual variables.

ASSIGNMENT

The assignment is interpreted by deleting all members of structures that do not appear in the by-name-parts-list, and reordering the reduced structures so that their members appear in order of the by-name-parts-list. The assignment is then carried out using these new structures.

For example:

```
DECLARE 1 CURRENTPRICE,  
        5 REGION1     FIXED DECIMAL (5,2),  
        5 REGION2     FIXED DECIMAL (5,2),  
        1 NEWPRICE,  
        5 REGION2     PICTURE '999V99',  
        5 REGION1     PICTURE '999V99';  
CURRENTPRICE = NEWPRICE;
```

In the assignment statement, NEWPRICE.REGION2 is assigned to CURRENTPRICE.REGION1 and NEWPRICE.REGION1 is assigned to CURRENTPRICE.REGION2. If the assignment statement had been

```
CURRENTPRICE = NEWPRICE, BY NAME;
```

NEWPRICE.REGION1 would have been assigned to CURRENTPRICE.REGION1 and NEWPRICE.REGION2 would have been assigned to CURRENTPRICE.REGION2.

The following example illustrates structure assignment using the BY NAME option:

```
DECLARE 1 BRAND1,  
        5 PRODUCT1,  
        10 RED     FIXED DECIMAL (5),  
        10 ORANGE  FIXED DECIMAL (5),  
        5 PRODUCT2,  
        10 YELLOW  FIXED DECIMAL (5),  
        10 BLUE    FIXED DECIMAL (5),  
        10 GREEN   FIXED DECIMAL (5);  
DECLARE 1 BRAND2,  
        5 PRODUCT1,  
        10 BLUE    FIXED DECIMAL (5),  
        10 GREEN   FIXED DECIMAL (5),  
        10 RED     FIXED DECIMAL (5),  
        5 PRODUCT2,  
        10 BROWN   FIXED DECIMAL (5),  
        10 YELLOW  FIXED DECIMAL (5);
```

The lists used to compose the by-name-parts-list are:

```

BRAND1:
  PRODUCT1:
    PRODUCT1.RED
    PRODUCT1.ORANGE
  PRODUCT2:
    PRODUCT2.YELLOW
    PRODUCT2.BLUE
    PRODUCT2.GREEN

```

```

BRAND2:
  PRODUCT1:
    PRODUCT1.BLUE
    PRODUCT1.GREEN
    PRODUCT1.RED
  PRODUCT2:
    PRODUCT2.BROWN
    PRODUCT2.YELLOW

```

If we code the assignment statement

```
BRAND1 = BRAND2, BY NAME;
```

the by-name-parts-list for the assignment is

```

PRODUCT1:
  PRODUCT1.RED
PRODUCT2:
  PRODUCT2.YELLOW

```

The assignment statement would therefore be the same as the following:

```

BRAND1.PRODUCT1.RED = BRAND2.PRODUCT1.RED;
BRAND1.PRODUCT2.YELLOW = BRAND2.PRODUCT2.YELLOW;

```

If we code the assignment statement

```
BRAND1.PRODUCT1 = BRAND2.PRODUCT1, BY NAME;
```

the by-names-parts-list for the assignment is

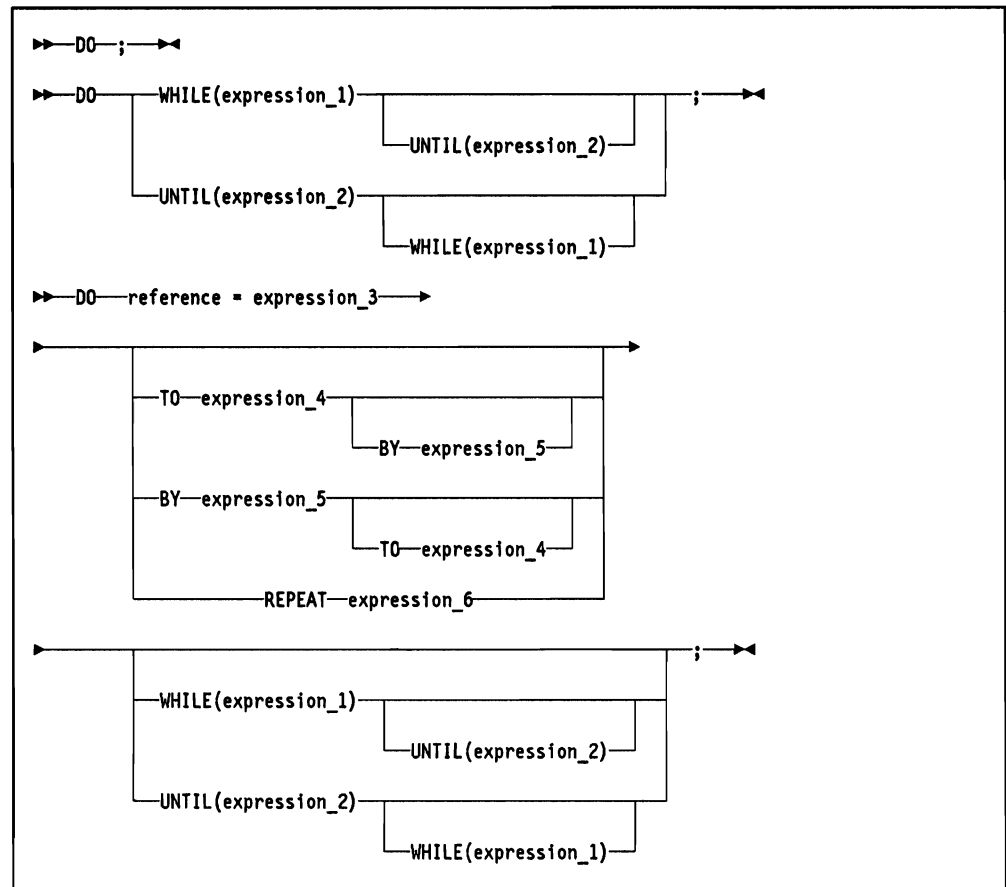
```
RED
```

DO Statement

The DO statement and its corresponding END statement delimit a group of statements collectively called a **do-group**. You can specify non-iterative or iterative processing of the group.

When the do-group ends normally, control passes to the next statement that can be processed, unless control leaves the do-group by a transfer of control, such as a GO TO, RETURN, or LEAVE statement.

DO



The type 1 DO statement specifies that the do-group, also called a simple do-group, is treated as a single statement. The do-group is processed once. It can be used, for example, to specify the THEN- or ELSE-unit of an IF statement. Types 2 and 3 provide for the iterative processing of the do-group.

WHILE(expression_1)

Evaluates to BIT(1). You can either use a bit variable that you have explicitly declared, or you can specify a condition, in which case you are implicitly declaring a bit variable which the program sets to '1'B or '0'B depending on whether the condition is true or false. Each time, before the do-group is processed, expression_1 is evaluated. If the value is '1'B, the do-group is processed. If the value is '0'B, processing of the do-group ends.

UNTIL(expression_2)

Must evaluate to BIT(1). You can either use a bit variable that you have explicitly declared, or you can specify a condition, in which case you are implicitly declaring a bit variable which the program sets to '1'B or '0'B depending on whether the condition is true or false. Each time, after the do-group is processed, expression_2 is evaluated. If the value is '0'B, the do-group is processed. If the value is '1'B, processing of the do-group ends.

reference

The **control variable**. It must be a scalar variable of arithmetic or pointer type. Expressions in the reference to the control variable, such as subscripts and

pointer qualifiers, are evaluated once, outside the do-group. Therefore, the location in storage of the control variable is established; it remains unchanged while the group is being processed.

If a reference is made to a control variable after the last iteration is completed, the value of the variable will be the value that exceeded the limit set in the DO statement.

expression_3

Specifies the initial value of the control variable. It must be of arithmetic, pointer or string type.

If the TO, BY or REPEAT options are omitted, the do-group is processed once with the control variable having the value of expression_3, or not at all if the WHILE option is specified and the value of expression_1 is '0'B.

TO expression_4

Specifies the ending value of the control variable. Expression_4 must be an arithmetic expression.

Processing of the do-group ends as soon as the value of the control variable is outside the range defined by the TO or REPEAT options. If the TO option is omitted, and the BY option is specified, processing is repeated until it is ended by a WHILE or UNTIL option, or until control is transferred out of the do-group.

BY expression_5

Specifies the increment added to the control variable after the do-group is processed. Expression_5 must be an arithmetic expression.

If the BY option is omitted, and the TO option is specified, expression_5 defaults to 1. If BY 0 is specified, processing is repeated until it is ended by a WHILE or UNTIL option, or until control is transferred out of the do-group.

REPEAT expression_6

Each time the do-group is processed, expression_6 is evaluated and assigned to the control variable. Processing is repeated until it is ended by the WHILE or UNTIL option, or until control is transferred out of the do-group.

Expression_6 must be of arithmetic, string, or pointer type.

The TO and BY options allow you to vary the control variable in fixed positive or negative increments. In contrast, the REPEAT option, an alternative to the TO and BY options, allows you to vary non-arithmetic control variables, such as pointers.

The effect of processing a do-group can be summarized as follows:

1. If a control variable is specified, the initial value is assigned to it and then any BY or TO options are evaluated. This yields the BY and TO values.
2. If the TO option is specified, the value of the control variable is tested against the TO value. The control variable is outside the range and the do-group is ended if:
 - The BY value is positive and the control variable is greater than the TO value.

DO

- The BY value is negative and the control variable is less than the TO value.
3. If the WHILE option is specified, expression_1 is evaluated. If the value is '0'B, the do-group is ended.
 4. The statements in the do-group are processed.
 5. If the UNTIL option is specified, expression_2 is evaluated. If the value is '1'B, the do-group is ended.
 6. If there is a control variable:
 - If the TO or BY option is specified, the BY value is added to the control variable.
 - If the REPEAT option is specified, expression_6 is evaluated and assigned to the control variable.
 - If the TO, BY, and REPEAT options are omitted, the do-group is ended.
 7. The cycle is repeated from point 2 if the do-group has not been previously ended.

Control can transfer into a type 1 do-group. It can transfer into a type 2 or type 3 do-group if the GOTO ends a procedure or an on-unit that was called from within the do-group.

The maximum depth of do-group nesting is 49.

Examples of DO Statements

The DO statement can specify a group of statements processed in the THEN clause or the ELSE clause of an IF statement, or in the WHEN statement or the OTHERWISE statement in a select-group. For example:

```
IF ITEM1 = ITEM2
  THEN DO;
      .
      .
      .
      END;
  ELSE DO INDEX1=1 TO 2;
      .
      .
      .
      END;
```

A repetitive do-group might take the form:

```
DO INDEX1 = 1 TO 10;
  .
  .
  .
  END;
```

In this example, the do-group is processed ten times, while the value of the control variable INDEX1 ranges from 1 through 10.

The following example specifies that the do-group is processed five times, with the value of INDEX1 equal to 2, 4, 6, 8, and 10:

```
DO INDEX1 = 2 TO 10 BY 2;
```

If negative increments of the control variable are required, the BY option must be used. For example:

```
DO INDEX1 = 10 TO 1 BY -1;
```

In the following example, the do-group is processed with INDEX1 equal to 1, 2, 4, 8, 16, and so on:

```
DO INDEX1 = 1 REPEAT 2*INDEX1;
.
.
.
END;
```

The WHILE and UNTIL options make successive processings of the do-group dependent upon a specified condition. For example:

```
DO WHILE (ITEM1 = ITEM2);
.
.
.
END;
DO UNTIL (ITEM1 = ITEM2);
.
.
.
END;
```

A crucial difference between DO WHILE and DO UNTIL is that DO WHILE checks at the beginning of each loop if the specified condition is true, but DO UNTIL makes this check at the end of the loop. The result is that in the absence of other options, a do-group headed by a DO UNTIL statement is processed at least once, but a do-group headed by a DO WHILE statement may not be processed at all. That is, the statements DO WHILE (A = B) and DO UNTIL (A \neq B) are not equivalent.

Consider the following examples:

```
DO WHILE (ITEM1 = ITEM2) UNTIL (ITEM3 = 10);
```

If ITEM1 is not equal to ITEM2 the first time the DO statement is processed, the do-group is not processed at all. If, however, ITEM1 is equal to ITEM2, the do-group is processed. If ITEM3 is equal to 10 after the do-group is processed, no further processing occurs. Process can occur again provided that ITEM3 is not equal to 10, and that ITEM1 is equal to ITEM2.

```
DO INDEX1 = 1 TO 10 UNTIL (ITEM1 = 1);
```

The do-group is processed at least once, with INDEX1 equal to 1. If ITEM1 is equal to 1 after the do-group is processed, no further processing occurs. Otherwise, the default increment (BY 1) is added to INDEX1, and the new value of INDEX1

END

is compared with 10. If INDEX1 is greater than 10, no further processing occurs. Otherwise, a new the do-group is processed again.

```
DO INDEX1 = 1 REPEAT (2 * INDEX1) UNTIL (INDEX1 = 256);
```

The first time the do-group is processed INDEX1 is equal to 1. After this, and each time the do-group is processed, the UNTIL expression is tested. If INDEX1 is equal to 256, no further processing occurs. Otherwise, the REPEAT expression is evaluated and assigned to INDEX1, and the do-loop is processed again.

```
DECLARE POINTER1          POINTER,  
        FIRST_ADDRESS    POINTER,  
        1 DATA_ITEM,  
        5 INTEGER        FIXED DECIMAL (7),  
        5 NEXT_ADDRESS   POINTER;  
  
.  
.  
DO POINTER1 = FIRST_ADDRESS  
  REPEAT POINTER1 -> DATA_ITEM  
  WHILE (POINTER1 != NULL());  
.  
.  
  POINTER1 = NEXT_ADDRESS;  
END;
```

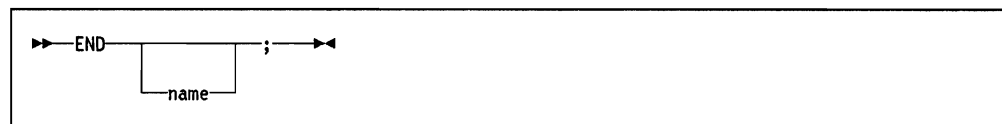
This example shows a DO statement used to step along a chained list, where each data item processed includes as one of its fields the address of the next item. The value FIRST_ADDRESS is assigned to POINTER1 for the first time the do-group is processed. Inside the loop, the address for the next item is assigned to POINTER1; and before each subsequent time the loop is processed, DATAITEM is defined as being based on POINTER1's current value. The last item in the chained list contains a null address, because there is no following item; therefore the value of POINTER1 is tested before the first and each time the do-group is processed; if it is NULL, no further processing occurs.

```
DO INDEX1 = 1 TO 10;  
  ARRAY1(INDEX1) = INDEX1;  
END;
```

This example shows how the control variable of a DO statement can be used as a subscript in statements within the do-group, so that each time processing occurs successive elements of a table or array are dealt with. This loop sets the first ten elements of ARRAY1 to 1,2,...,10 respectively.

END Statement

The END statement and the corresponding PROCEDURE, BEGIN, DO or SELECT statement delimit blocks and do-groups.



name

An identifier that designates a statement label or entry constant.

Each END statement must pair with a PROCEDURE, BEGIN, DO or SELECT statement. Pairing is processed outwards from the innermost procedure, begin-block, or do-group. For example:

```
PROC1: PROCEDURE;
    BEGIN1: BEGIN;
        D01: DO;
            END;
        END;
    END;
```

In this example, the innermost END statement is paired with the DO statement; the next END statement, working outwards, is paired with the BEGIN statement; then the next END statement is paired with the PROCEDURE statement.

If END is followed by a name, the name must match that of the PROCEDURE, BEGIN, DO, or SELECT statement with which it is paired. For example, the END statement that is paired with BEGIN1: BEGIN could be written as

```
END BEGIN1;
```

A program ends normally when control reaches the END statement of the first procedure called in the program. The program would also end normally if control reached a RETURN statement in the first procedure; but RETURN is not usually coded in a program's first procedure.

Processing of a procedure or begin-block ends normally when control reaches the END statement for the block.

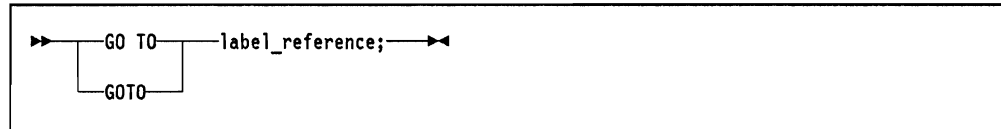
Processing of a do-group ends when control reaches the END statement of the group for the final time, in accordance with the conditions specified in the DO statement.

Processing of a function ends abnormally if control reaches the END statement; a RETURN statement must be specified. (See "RETURN Statement" on page 14-4 for more information.)

GO TO Statement

The GO TO statement transfers control to the statement identified by the label-reference.

GO TO



label_reference

Specifies a label constant, a label variable, or function reference that returns a label value. Because a label variable or function reference can have different values at time the GO TO statement is processed, control may not always pass to the same statement.

When a GO TO statement transfers control out of a function, the evaluation of the expression that contained the corresponding function reference is discontinued.

Transferring control out of a block using a GO TO statement can sometimes result in the ending of several procedures and/or begin-blocks. Specifically, if the transfer point specified by the GO TO statement is contained in a block that did not directly activate the block being ended, all intervening blocks in the activation sequence are ended (see "Ending a Procedure" on page 4-11 for details). For example:

```
PROC1: PROCEDURE;
.
.
  BEGIN1: BEGIN;
.
.
  CALL PROC2;
.
  END BEGIN1;
.
.
  PROC2: PROCEDURE;
.
.
.
  BEGIN2: BEGIN;
.
.
  GO TO OUTSIDE;
.
  END BEGIN2;
.
  END PROC2;
.
  OUTSIDE: ITEM = 0;
.
  END PROC1;
```

In the above example, PROC1 activates BEGIN1, which activates PROC2, which activates BEGIN2. In BEGIN2, the statement GO TO OUTSIDE transfers control to the statement in PROC1 that is labeled OUTSIDE. Because this statement is not contained in BEGIN2, PROC2, or BEGIN1, all three blocks are ended; PROC1 remains active. Therefore, the transfer of control out of BEGIN2 results in

the ending of intervening blocks BEGIN1 and PROC2 as well as the ending of block BEGIN2.

A GO TO statement cannot transfer control to an inactive block. It cannot transfer control from outside a do-group to a statement inside a type 2 or type 3 do-group, unless the GO TO ends a procedure or on-unit called from within the do-group. For definitions of type 2 and type 3 do-groups, see "DO Statement" on page 13-5.

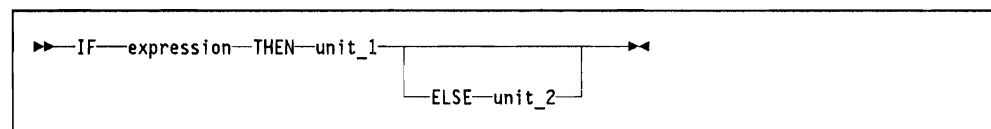
If you use a label variable or a function reference in a GO TO statement, its value must be a valid label, and the block containing the assigned label constant must be active.

The GO TO statement specifies an unconditional transfer of control. If the destination of the GO TO is specified by a label variable or a function reference, you can use it as a switch.

Use of the GO TO statement leads to difficulty in debugging and maintaining programs because the permanent transfers of control it creates within the program are hard to trace. Whenever possible, you should use do-groups and procedures instead of GO TO statements.

IF Statement

The IF statement tests the value of an expression and controls the flow of processing according to the result of that test.



expression

Evaluates to BIT(1). If the expression is a condition such as INTEGER1 = 0, the program evaluates the expression and sets the bit to 1 if the expression is true, and to 0 if the expression is false. Alternatively, the expression can be a BIT(1) variable which the program has set to 1 or 0. For instance, you can define EOF as a BIT(1) variable with an initial value of 0, and set it to 1 when end of file is detected and handled by an ENDFILE on-unit.

unit

Each unit is either a single statement (except BEGIN, DECLARE, DO, END, PROCEDURE, SELECT, or a directive), a do-group, or a begin block. Each unit can contain statements that transfer control (such as GO TO), so that the normal sequence of the IF statement can be overridden.

The IF statement is a compound statement. The semicolon ending the last unit also ends the IF statement.

If the expression evaluates to '1'B, unit_1 is processed and unit_2 is ignored. If the expression evaluates to '0'B, unit_1 is ignored and unit_2 is processed, if present.

ITERATE

You can nest IF statements by specifying either or both units as IF statements. Because each ELSE unit is associated with the innermost unmatched IF in the same block or do-group, you may need to specify an ELSE with a null statement to achieve a desired sequence of control.

The maximum depth of IF statement nesting is 49.

Examples of IF Statements

```
IF ITEM1 > ITEM2
  THEN LARGERNO = ITEM1;
  ELSE LARGERNO = ITEM2;
```

In this example, if the value of ITEM1 is greater than the value of ITEM2, the value of ITEM1 is assigned to LARGERNO, and the ELSE unit is not processed. If the value of ITEM1 is less than or equal to the value of ITEM2, the THEN unit is not processed, and the value of ITEM2 is assigned to LARGERNO.

You do not always have to specify an ELSE unit. When in the event that the IF condition is false you simply want to pass control to the statement following the IF statement, you can omit the ELSE unit. For example:

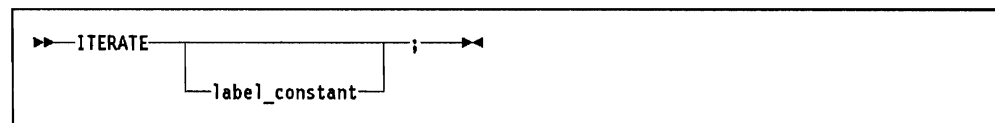
```
      IF (ITEM1 = ITEM2) & (ITEM3 = ITEM4)
        THEN CALL PROC1;
NEXT: ITEM3 = ITEM1 + ITEM2;
```

If the expression is true, the CALL statement of the THEN unit calls PROC1 for processing. If the expression is false, the THEN unit is not processed and control passes directly to the statement labeled NEXT.

IBM Extension

ITERATE Statement

The ITERATE statement is valid only within an iterative do-group. The ITERATE statement transfers control to the END statement that delimits the iterative do-group.



label_constant

Must be a label of a containing do-group. If label_constant is omitted, control is transferred to the END statement of the do-group immediately containing the ITERATE statement.

Example of the ITERATE Statement

```

QCOUNT: PROCEDURE OPTIONS (MAIN);
    DECLARE SUBJECTSTRING CHARACTER (80),
           STRINGLENGTH  FIXED BINARY (15),
           INDEX1        FIXED BINARY (15),
           INDEX2        FIXED BINARY (15);
    GET EDIT (SUBJECTSTRING) (A(80));
    STRINGLENGTH = 80;

    /* Remove all Q's from SUBJECTSTRING */

MAINLOOP: DO INDEX1 = 1
    REPEAT (INDEX1 + 1)
    UNTIL (INDEX1 = STRINGLENGTH);

    IF SUBSTR (SUBJECTSTRING,INDEX1,1) = 'Q'
    THEN ITERATE MAINLOOP;

    /* Replace Q with following character; shift */
    /* remaining characters left accordingly */

    DO INDEX2 = INDEX1 TO (STRINGLENGTH - 1);
    SUBSTR (SUBJECTSTRING,INDEX2,1)
    = SUBSTR (SUBJECTSTRING,INDEX2+1,1);
    END;

    /* String has lost one character */

    STRINGLENGTH = STRINGLENGTH - 1;

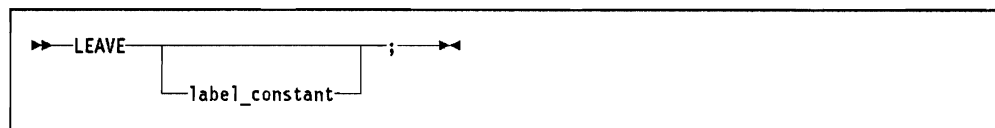
    /* Check character replacing deleted Q */

    INDEX1 = INDEX1 - 1;
    END MAINLOOP;
END QCOUNT;

```

LEAVE Statement

The LEAVE statement transfers control from within a do-group to the statement following the END statement that delimits the group and ends the do-group. LEAVE is valid only within a do-group.



label_constant

Must be a label of a containing do-group. The do-group that is left is the do-group that has the specified label. If label_constant is omitted, the do-group that is left is the group that contains the LEAVE statement.

LEAVE

The LEAVE statement and the referenced or implied DO statement must not be in different blocks.

Examples of LEAVE Statements

In the following example, the LEAVE statement transfers control to “next statement”:

```
DO . . . ;  
.  
.  
  LEAVE;  
.  
.  
END;  
next statement;
```

In the following example, the statement LEAVE GROUP1 transfers control to “statement after GROUP1”:

```
GROUP1: DO INDEX1 = 1 TO 10;  
        DO INDEX2 = 1 TO 5;  
          IF SAMPLEARRAY(INDEX1,INDEX2) = 0  
            THEN LEAVE GROUP1;  
          ELSE  
            SAMPLEARRAY(INDEX1,INDEX2) = 5;  
          END;  
        statement within GROUP1;  
      END GROUP1;  
statement after GROUP1;
```

In the following example, LEAVE MAINLOOP causes processing of MAINLOOP to end once the letter Q has been found in SUBJECTSTRING:

```

        DECLARE SUBJECTSTRING  CHARACTER (80),
               INDEX1          FIXED BINARY (15),
               QSTRINGTOTAL    FIXED BINARY (15),
               QPRESENT        FIXED BINARY (15)
                               STATIC INITIAL (0);

        .
        .
        .
        GET EDIT (SUBJECTSTRING) (A(80));

        /* Determine if Q is present in string */

MAINLOOP: DO INDEX1 = 1 TO 80;
           IF SUBSTR (SUBJECTSTRING,INDEX1,1) = 'Q'
               THEN DO;
                   QPRESENT = 1;
                   LEAVE MAINLOOP;
               END;
        END MAINLOOP;

        /* Accumulation of total number */
        /* of strings containing Q */

        EXIT: QSTRINGTOTAL = QSTRINGTOTAL + QPRESENT;

```

In this example, if no Qs are present in SUBJECTSTRING, MAINLOOP is processed 80 times, and QPRESENT is left with a value of zero, so that QSTRINGTOTAL remains unchanged after QPRESENT is added. If a Q is present in the string, QPRESENT is set to 1, and the LEAVE statement is processed, so that MAINLOOP ends and control is transferred to EXIT.

End of IBM Extension

Null Statement

The null statement causes no action and does not affect sequential processing.

```

▶▶;▶▶

```

If a statement is preceded by a labeled null statement, a GOTO to that label is effectively a transfer of control to the following statement, even if that statement cannot itself be labeled.

Examples of Null Statements

The null statement can specify that no action is taken when a condition is raised. For example:

```

ON ENDPAGE(SAMPLEFILE);

```

In this example, no action is taken when the ENDPAGE condition is raised for file SAMPLEFILE.

SELECT, WHEN, OTHERWISE

A null statement can specify that no action is taken in the THEN unit of an IF statement. For example:

```
IF ITEM1 = ITEM2
  THEN;
  ELSE ITEM1 = 15;
NEXT: ITEM2 = 25;
```

In this example, if ITEM1 is equal to ITEM2, control passes to NEXT. If ITEM1 is not equal to ITEM2, the ELSE unit is processed before control passes to NEXT.

A null statement can similarly specify that no action is to be taken in the ELSE unit of an IF statement. For example:

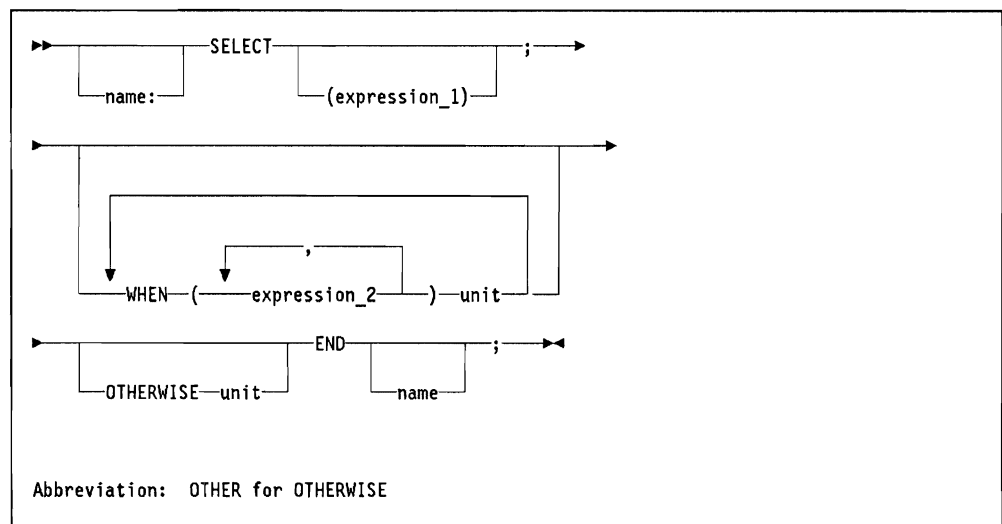
```
IF ITEM1 = ITEM2
  THEN ITEM1 = 15;
  ELSE;
NEXT: ITEM2 = 25;
```

In this example, if ITEM1 is equal to ITEM2, the value 15 is assigned to ITEM1. If, however, ITEM1 is not equal to ITEM2, control passes directly to NEXT.

IBM Extension

SELECT, WHEN, and OTHERWISE Statements

A select-group provides a multi-way conditional branch. A select-group contains a SELECT statement, optionally one or more WHEN statements, optionally an OTHERWISE statement, and an END statement. The syntax of the select-group is shown below:



name: SELECT (expression_1);

With its corresponding END statement, SELECT delimits a group of statements, collectively called a select-group. The optional expression_1 in the SELECT statement is evaluated and its value is saved.

WHEN (expression_2) unit

Specifies an expression or expressions that are evaluated and compared one by one with the saved value of expression_1. If an expression is found equal, the evaluation of the expressions in the WHEN statements is ended, and the unit of the associated WHEN statement is processed. If no such expression is found in the WHEN statements, the next statement is processed.

If expression_1 is omitted, each expression_2 must evaluate to a bit string. If the result is '1'B, the unit of the associated WHEN statement is processed. If the result is '0'B, the next WHEN statement is evaluated. If all WHEN statements evaluate to '0'B, the unit of the OTHERWISE statement is processed.

The WHEN statement must not have a label prefix.

OTHERWISE unit

Specifies the unit is processed when every test of the preceding WHEN statements fails.

If the OTHERWISE statement is omitted and processing of the select-group does not result in the selection of a unit, the ERROR condition is raised.

The OTHERWISE statement must not have a label prefix.

unit

Each unit is either a single statement (except BEGIN, DECLARE, DO, END, PROCEDURE, SELECT, or a directive), a do-group, or a begin-block. Each unit can contain statements that transfer control (such as GO TO); hence, the normal sequence of the SELECT statement can be overridden.

Each unit may be labeled.

END name;

Must be specified. It ends the select-group (see "END Statement" on page 13-10).

After processing of a unit of a WHEN or OTHERWISE statement, control passes to the statement following the select-group, unless the normal flow of control is altered within the unit.

The maximum permissible depth of nesting of select-groups is 49.

Examples of Select-Groups

In the following example, E, E1, etc., are expressions. When control reaches the SELECT statement, the expression E is evaluated and its value is saved. The expressions in the WHEN statements are then evaluated in turn (in the order in which they appear), and each value is compared with the value of E. If a value is found that is equal to the value of E, the action following the corresponding WHEN statement is processed; no further WHEN statement expressions are evaluated. If none of the expressions in the WHEN statements is equal to the expression in the

STOP

SELECT statement, the action specified by the OTHERWISE statement is processed.

```
SELECT (E);
  WHEN (E1,E2,E3) action-1;
  WHEN (E4,E5) action-2;
  OTHERWISE action-n;
END;
NL: next statement;
```

An example of expression_1 being omitted is:

```
SELECT;
  WHEN (ITEM1>ITEM2) CALL BIGGER;
  WHEN (A=B) CALL SAME;
  OTHERWISE CALL SMALLER;
END;
```

If a select-group contains no WHEN statements, the action in the OTHERWISE statement is processed unconditionally. If the OTHERWISE statement is omitted, and processing of the select-group does not result in the selection of a WHEN statement, the ERROR condition is raised.

End of IBM Extension

STOP Statement

The STOP statement abnormally ends the run unit.

```
▶▶STOP;◀◀
```

When you process the STOP statement, any files in the run unit that are open are closed with an error indication.

Chapter 14. Procedures, Subroutines, and Functions

You may write your own subroutines and functions (user-defined), or use those provided by the PL/I compiler (built-in).

This chapter describes user-defined subroutines and user-defined functions, and how to define, declare, and call them. When not stated to the contrary, references to “subroutine” and “function” in this chapter are to user-defined procedures. For details about built-in functions and subroutines, see Chapter 15, “Built-In Functions, Subroutines, and Pseudovariables.”

Subroutines and functions can:

- Be called from different points in a program as well as in different programs to process the same frequently used process.
- Process data passed to them on different calls.
- In the case of subroutines, return control to a point immediately following the point of calling or transfer control to another part of the program.
- In the case of functions, return control and a value to the point of calling or transfer control to another part of the program.

Subroutines and functions can use data known in the calling block and made available by:

- Arguments and parameters. References to data in the calling block are passed in an argument list to parameters in the called procedure.
- Names whose scope of declaration includes both the calling block and the called procedure (see “Scopes of Names” on page 4-14).

Defining a Procedure

You define a procedure by writing a PROCEDURE statement as the first of a sequence of statements and an END statement as the last. (See “Internal and External Procedures” on page 4-9 for a discussion of procedures, and “END Statement” on page 13-10 for a discussion of the END statement.)

Functions

A **function** is a procedure that has a RETURNS option in the PROCEDURE statement. A function is called by a function reference. It ends normally by processing a RETURN(expression) statement and returning a scalar value to the point of calling.

DEFINING A PROCEDURE

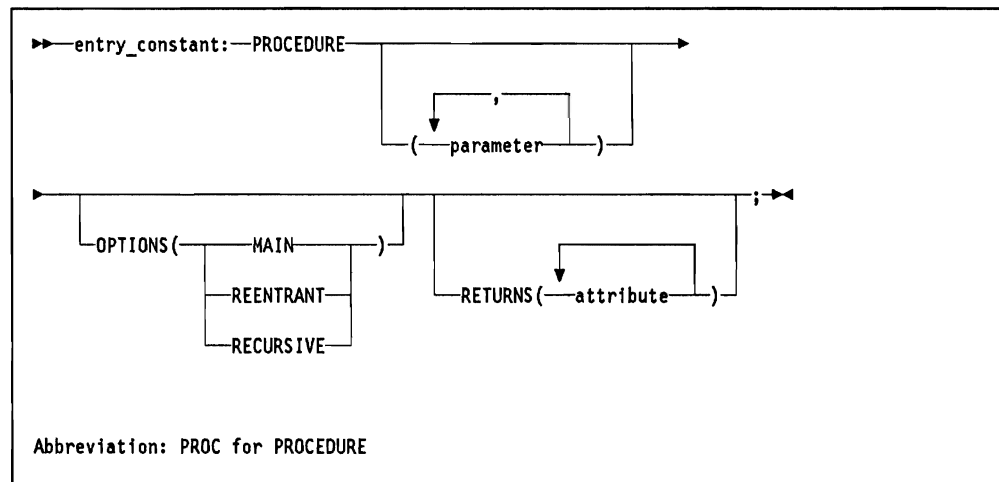
Subroutines

A subroutine is a procedure that has no RETURNS option in the PROCEDURE statement. A subroutine is called by a subroutine CALL. It ends normally by processing either a RETURN statement that has no expression or an END statement.

PROCEDURE Statement

The PROCEDURE statement specifies:

- Any parameters the procedure may have.
- For an external procedure, any options the procedure may have.
- For a function, the attributes of the returned value.



entry_constant

A name that is the entry name of the procedure. Every procedure must have an entry name. The maximum length of an external entry name is 10 characters.

parameter

A name that is used to refer to an argument passed to this procedure. It is explicitly declared as a parameter by its appearance in a parameter list. See "Parameter Attributes" on page 14-3 for how to declare attributes for parameters.

The maximum number of parameters any procedure can have is 32, except for function procedures, which can have a maximum of 31.

IBM Extension

A parameter list can be specified for the external procedure. See "Calling a PL/I Program from a Non-PL/I Program" on page 2-24.

End of IBM Extension

OPTIONS

OPTIONS is valid only for an external procedure. If you specify OPTIONS, you must include at least one option.

The options are separated by blanks or commas.

The options are syntax checked but otherwise ignored. They are provided for easy transfer of procedures to other compilers that require them. Although they are not required for AS/400 PL/I, you may include them so that your programs can be used with other compilers.

MAIN

The MAIN option, if specified, is syntax checked but otherwise ignored.

The usual first statement of a PL/I program is

```
Label: PROCEDURE OPTIONS (MAIN);
```

REENTRANT

The REENTRANT option, if specified, is syntax checked but otherwise ignored. All AS/400 PL/I procedures are reentrant.

RECURSIVE

The RECURSIVE option, if specified, is syntax checked but otherwise ignored. All AS/400 PL/I procedures are recursive. For more information on using recursive procedures, refer to "Recursive Procedures" on page 14-11.

RETURNS(attribute)

The value of the expression in a RETURN statement is converted to the attributes specified in the RETURNS option before being returned.

The attributes can specify any scalar data type except FILE. String lengths must be specified by integer constants. If you specify the ENTRY attribute, it must not have parameter descriptors or a RETURNS attribute.

Parameter Attributes

You supply the attributes of a parameter in a DECLARE statement internal to the procedure.

For a parameter or component of a parameter structure that is scalar or an array of scalars, you must specify at least one of the following attributes, unless you use * to pass the parameter to a non-PL/I routine:

| | |
|---------|-----------|
| FIXED | CHARACTER |
| FLOAT | BIT |
| BINARY | POINTER |
| DECIMAL | LABEL |
| PICTURE | ENTRY |

You cannot specify storage class or scope attributes for a parameter.

CALLING A PROCEDURE

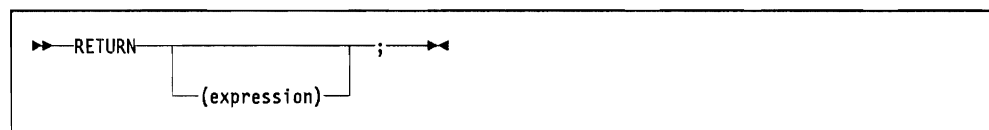
Parameter Lengths and Bounds

Lengths of strings and bounds of arrays must be integer constants, or they must be specified by means of the asterisk notation (see "BIT and CHARACTER Attributes" on page 12-16 and "Arrays and the Dimension Attribute" on page 12-38).

If you specify the string length or an array extent of a parameter by means of an asterisk, the length or bounds will be taken from the associated argument. This is useful if argument lengths or bounds differ for different calls or if their values are known only during processing.

RETURN Statement

The RETURN statement ends the procedure that contains it and returns control to the calling procedure. In the case of a function, it also returns a value.



expression

A scalar expression of any data type except FILE.

A RETURN statement without an expression ends a subroutine. Control returns to the next statement following the point of calling.

A RETURN statement with an expression ends the processing of a function. The expression must have a type that can be converted to the attributes specified in the RETURNS option of the procedure statement; that is, the attributes in the RETURNS option and the attributes of the expression must both specify a problem data type or they must both specify the same program control data type.

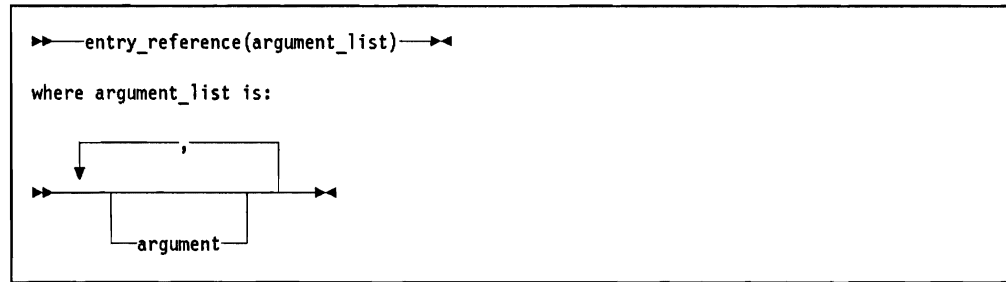
The expression is evaluated and converted to the attributes of the RETURNS option. The resulting value is returned to the point of calling.

Calling a Procedure

A function is called by a function reference; a subroutine is called by a subroutine call. Both subroutines and functions may be nested up to 50 levels deep. (See "Function Reference" below and "CALL Statement" on page 14-7.)

Function Reference

A **function reference** is an entry constant or an entry variable reference followed by a possibly empty argument list. The entry constant or entry variable must represent a function.



You can write a function reference wherever an expression is allowed as well as in a subroutine call if the function returns an entry value.

To call a function that has no arguments, specify the function name with an empty argument list. Like a subroutine, a function can operate upon the arguments passed to it and upon other known data.

When a function ends normally, by means of a RETURN(expression) statement, the value determined by the function is converted, if necessary, to the attributes of the RETURNS option of the PROCEDURE statement. The value is then returned, with control, to the point of calling. Evaluation of the expression then continues.

The returned value can be any type of scalar data item except a file.

The examples that follow show entry constants and entry variables used in function references. In these examples, the function references call functions that are internal to the initial procedure.

The following example shows an entry constant used in a function reference:

```

SKYE: PROCEDURE;
  DECLARE (A,B,C,D) FIXED BINARY (15);
  . . .
  A = FUN(C,D)*B;
  FUN: PROCEDURE(Q,R)
    RETURNS (FIXED BINARY (15));
    DECLARE (Q,R) FIXED BINARY (15);
    RETURN (3.1416*Q*R);
  END FUN;
END SKYE;

```

In this example, the function reference FUN(C,D) calls the function FUN, with two arguments. Argument C is associated with parameter Q, and argument D with parameter R. FUN returns the value of 3.1416*Q*R to the function reference. The returned value is then multiplied by the value of B, and the result is assigned to A.

The following example shows an entry variable used in a function reference:

CALLING A PROCEDURE

```
EIGG: PROCEDURE;
      DECLARE (A,B,C,D,E) FIXED BINARY (15),
              FUN          ENTRY (FIXED BINARY (15))
                          VARIABLE
                          RETURNS(FIXED BINARY (15));
      . . .
      IF A>B THEN FUN = FUN_1;
              ELSE FUN = FUN_2;
      C = D*FUN(E);
      FUN_1: PROCEDURE (Q)
              RETURNS (FIXED BINARY (15));
              DECLARE Q FIXED BINARY (15);
              RETURN (3.1416*Q**2);
      END FUN_1;
      FUN_2: PROCEDURE(Q)
              RETURNS(FIXED BINARY (15));
              DECLARE Q FIXED BINARY (15);
              RETURN (Q**2);
      END FUN_2;
      END EIGG;
```

In this example, the entry constant FUN_1 or FUN_2 is assigned to the entry variable FUN, after the comparison of A and B. The function reference FUN(E) calls the appropriate function it has a single argument. Argument E is associated with parameter Q. The called function returns a value to the function reference. The returned value is then multiplied by the value of D and the result is assigned to C.

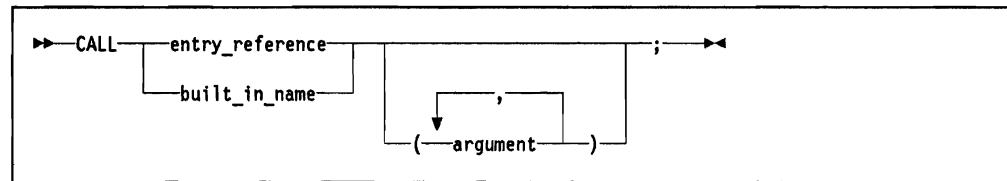
The following example shows a pointer-qualified and subscripted entry variable reference used in a function reference:

```
CARA: PROCEDURE;
      DECLARE 1 A(10) BASED,
              5 B FLOAT BINARY (5),
              5 C ENTRY(FIXED DECIMAL (6))
              RETURNS (FIXED BINARY (15)),
              5 D FIXED DECIMAL (6);
      DECLARE P POINTER;
      DECLARE I FIXED BINARY (15);
      . . .
      P->C(3) = FUN;
      I = P->C(3)(13);
      FUN: PROCEDURE (Q) RETURNS (FIXED BINARY(15));
              DECLARE Q FIXED DECIMAL (6),
                      RI FIXED BINARY(15);
              . . .
              RETURN (RI);
      END FUN;
      . . .
      END CARA;
```

In this example, the entry constant FUN is assigned to element C(3) of the array of structures A. The function reference C(3)(13) calls function FUN; it has a single argument. The argument, 13, is associated with parameter Q. FUN returns a binary fixed-point value, which is then assigned to variable I.

CALL Statement

You use the CALL statement to call a subroutine, if it is user-defined or built-in.



entry_reference

An entry constant, an entry variable reference, or a function reference that returns an entry value. In each case, the value of the entry reference must represent a subroutine.

built_in_name

The name of a built-in subroutine (see “Built-In Subroutines” on page 15-4).

argument

A scalar expression or an array or structure reference, which is evaluated in the procedure in which the call is processed. There can be no more than 32 arguments in the argument list. For a function call, there is a maximum of 31 arguments, because the RETURNS value is implicitly the first argument.

If the entry reference is a function reference, it must return a subroutine that does not itself have parameters. This is illustrated in the third example below (procedure IONA).

A **subroutine call** is an entry reference followed by an optional and possibly empty argument list that appears in a CALL statement. The entry reference must represent a subroutine.

Whenever a subroutine is called, any arguments of the calling statement are associated with the parameters of the procedure (see “Association of Arguments and Parameters” on page 14-9). Control is then passed to that procedure. The subroutine is therefore activated and processing begins.

The examples that follow show entry constants, entry variables, and function references used in subroutine calls, as well as how a subroutine interacts with the procedure that calls it. In these examples, subroutines are called that are internal to the initial procedure.

The following example shows an entry constant used in a subroutine call:

CALLING A PROCEDURE

```
GIGHA: PROCEDURE;  
  DECLARE (A,B) FIXED BINARY (15);  
  . . .  
  CALL SUB (A);  
  B=B+1;  
  . . .  
  SUB: PROCEDURE(C);  
    DECLARE C FIXED BINARY (15);  
    . . .  
    IF C < 12 THEN RETURN;  
    . . .  
  END SUB;  
END GIGHA;
```

In this example, the subroutine call `CALL SUB(A)` calls the internal subroutine `SUB`. The argument `A` in the subroutine call is associated with parameter `C` of the procedure `SUB`. Each reference to `C` in `SUB` is treated as a reference to `A`.

When `SUB` ends, control passes to the first processable statement following the subroutine call; that is, to the statement `B = B + 1`.

The following example shows an entry variable, which references a subroutine, used in a subroutine call:

```
ISLAY: PROCEDURE;  
  DECLARE SUB ENTRY (FIXED BINARY (15))  
    VARIABLE,  
    A FIXED BINARY (15);  
  . . .  
  IF A>11 THEN SUB=SUB1;  
    ELSE SUB=SUB2;  
  CALL SUB (A);  
  SUB1: PROCEDURE (B);  
    DECLARE B FIXED BINARY (15);  
    . . .  
    IF B = 12 THEN RETURN;  
    . . .  
  END SUB1;  
  SUB2: PROCEDURE (C);  
    DECLARE C FIXED BINARY (15);  
    . . .  
    IF C > 7 THEN RETURN;  
    . . .  
  END SUB2;  
END ISLAY;
```

In this example, the internal entry constant `SUB1` or `SUB2` is assigned to the entry variable `SUB`. The subroutine call `CALL SUB(A)` calls the internal subroutine `SUB1` or `SUB2` as appropriate. The argument `A` in the subroutine call is associated with parameter `B` in `SUB1` or parameter `C` in `SUB2`. Each reference to `B` in `SUB1` or to `C` in `SUB2` is treated as a reference to `A`.

When the called subroutine ends, control passes to the first statement that can be processed following the subroutine call; that is, to the statement `END ISLAY`.

The following example shows an entry value that is returned by a function and called immediately in a subroutine call:

```

IONA: PROCEDURE;
      . . .
      CALL FUN(13)();
FUN:  PROCEDURE(A) RETURNS (ENTRY);
      DECLARE A FIXED DECIMAL (6,2);
      . . .
      IF A>9 THEN RETURN (SUB1);
      ELSE RETURN (SUB2);
      END FUN;
SUB1: PROCEDURE;
      . . .
      END SUB1;
SUB2: PROCEDURE;
      . . .
      END SUB2;
STMT1: A = 5;
END IONA;
    
```

In this example, the function reference FUN(13) calls the internal function FUN. The argument 13 in the function reference is associated with parameter A of the function FUN. A dummy argument is created because the argument is a constant.

When FUN ends, control is returned to the subroutine call together with an entry value (SUB1 in this example), which is then called as a subroutine without parameters.

When the subroutine call CALL FUN(13)() ends, control passes to the first processable statement following the subroutine call (STMT1 in this example).

Association of Arguments and Parameters

When a function or subroutine is called, parameters in the parameter list correspond, from left to right, to arguments in the associated argument list. The number of arguments and parameters must be the same for a PL/I procedure.

IBM Extension

For an external procedure declared with the ASSEMBLER option, or for a built-in subroutine, any number of arguments, from zero through to the number of parameters, can be passed in the call that calls the routine, but intervening arguments cannot be omitted. Note that the ASSEMBLER option means any module that can be processed. For example, if routine SUB3 has three parameters, the following may be valid:

```
CALL SUB3(AFID,BFIB);
```

However, the following is not valid:

```
CALL SUB3(AFIB,,BFL0D);
```

End of IBM Extension

CALLING A PROCEDURE

There is no restriction on the data type of problem data arguments that can be passed to problem data parameters. The data type of program control data arguments must be the same as the program control data parameters to which they are passed.

An argument can be associated with a parameter in any of the following ways:

- By passing a reference to the argument, rather than its value. This is done when the argument is a variable whose attributes match those of the corresponding parameter, as described below. Any change to the value of the parameter will affect the value or the original argument.

An easy way to force the creation of a dummy argument is to enclose the reference in parentheses, thereby turning it into an expression.

- By creating a **dummy argument** to which the value of the argument is assigned and passing the dummy argument. This is done in the remaining cases; that is, when the argument is:
 - a constant
 - an expression with operators or parentheses
 - a function-reference
 - a variable whose attributes do not match those of the parameter.

The argument is converted, if necessary, to the attributes of the parameter before being assigned to the dummy argument. The dummy argument and the parameter initially have the same value as the original argument, but any change made to the value of the parameter affects only the value of the dummy argument. The value of the original argument is unchanged. A reference to the parameter is a reference to its dummy argument.

The parameter attributes used for dummy argument creation and conversion are:

- Those declared for the corresponding parameter, in the case of an internal procedure
- Those specified in the corresponding parameter descriptor of the ENTRY attribute, in the case of an external procedure or entry variable.

The argument and the parameter are considered to match if they have the same data and alignment attributes. If a parameter string length or array bound is specified by an integer constant, the corresponding length or bound of the argument must be an integer constant with the same value. If a parameter string length or array extent is specified by an asterisk, the corresponding length or bounds of the argument may have any value.

If a parameter is a scalar, the argument must be scalar.

If a parameter is an aggregate, the argument must be an aggregate with identical size and shape and identical component data types. If an array bound of the parameter is specified by an integer constant, the corresponding array bound of the argument must be specified by an integer constant with the same value. Therefore, a dummy argument will never be created for an aggregate.

The rules that govern the creation of a dummy argument for a non-PL/I routine are the same as those for a PL/I procedure, except that no dummy argument is created for a non-PL/I routine in the following cases:

- When a variable is passed to a parameter that is described by an asterisk.
- When the argument is a file constant.

For more information about passing arguments to non-PL/I routines, see “OPTIONS(ASSEMBLER) Attribute” on page 12-36.

Recursive Procedures

An active procedure that can be called from within itself or from within another active procedure is said to be a **recursive procedure**; such a call is termed recursion.

A procedure that is called recursively should have the **RECURSIVE** option specified in its **PROCEDURE** statement for compatibility with other implementations of PL/I.

The environment (that is, values of automatic variables, etc.) of every call of a recursive procedure is “pushed down” at a recursive call, and “popped up” at the end of that call. A label constant in the current block is always a reference to the current calling of the block that contains the label.

If a label constant is assigned to a label variable in a particular calling, a **GO TO** statement naming that variable in another call would restore the environment that existed when the assignment was processed.

The environment of a procedure called from within a recursive procedure by means of an entry variable is the one that was current when the entry constant was assigned to the variable. Consider the following example:

```

I=1;
CALL A;    /*FIRST CALLING OF A*/

A: PROCEDURE OPTIONS(RECURSIVE);
  DECLARE EV ENTRY VARIABLE STATIC;
  IF I=1 THEN
    DO;
      I=2;
      EV=B;
      CALL A;    /*2ND CALLING OF A*/
    END;
  ELSE CALL EV; /*CALLS B WITH
                ENVIRONMENT OF FIRST
                CALLING OF A*/
B: PROCEDURE;
  GO TO OUT;
  END B;
OUT: END A;

```

The **GO TO** statement in the procedure B will transfer control to the **END A** statement in the first calling of A, and will therefore end B and both calls of A.

CALLING A PROCEDURE

Effect of Recursion on Automatic Variables

The values of variables allocated in one activation of a recursive procedure must be protected from change by other activations. A stack operates on a last-in first-out basis. The most recent generation of an automatic variable is the only one that can be referenced. Static variables are not affected by recursion. Therefore they are useful for communication across recursive calls. This also applies to based variables and to automatic variables that are declared in a procedure that contains a recursive procedure. For example:

```
A: PROCEDURE;  
  DECLARE X ... ;  
  .  
  .  
  .  
  B: PROCEDURE OPTIONS(RECURSIVE);  
    DECLARE Z ... ,  
      Y STATIC;  
    CALL B;  
    .  
    .  
    .  
  END B;  
END A;
```

A single generation of the variable X exists throughout calls of procedure B. The variable Z will have a different generation for each call of procedure B. The variable Y can be referred to only in procedure B and will not be reallocated at each call.

Chapter 15. Built-In Functions, Subroutines, and Pseudovariables

The built-in functions, subroutines and pseudovariables are described in alphabetical order later in this chapter. In general, each description has the following:

- The syntax of the reference
- A description of the value returned by the built-in function or the target identified by the pseudovariable
- Details of the arguments
- Any other qualifications on the use of the function, subroutine or pseudovariable.

The arguments may be expressions. The arguments are evaluated, checked for correct attributes, and converted if necessary to a form suitable for the built-in function, subroutine, or pseudovariable according to the rules for data conversion.

Arguments must be scalar, except for those that indicate that they accept aggregates or structures (see “Aggregate Arguments” on page 15-5).

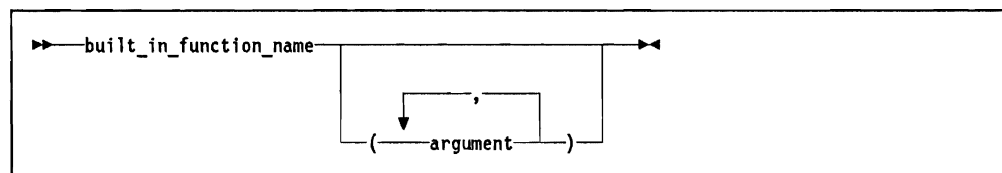
Declaring a Built-In Function or Built-In Subroutine

A built-in function or built-in subroutine can be declared explicitly or contextually. They are explicitly declared in a DECLARE statement by the BUILTIN attribute. A built-in function is contextually declared by a built-in function reference that contains an argument list; a built-in subroutine is contextually declared by a built-in subroutine call with an optional argument list.

Some built-in functions have two names: a full name and an abbreviated name. Each has a separate declaration (explicit or contextual) and name scope, like any two other names. Although both refer to the same built-in function, you can use both names in built-in function references in the same block.

Built-In Functions

A **built-in function** is a predefined function that is called by a built-in function reference. A **built-in function reference** is a built-in function name with an optional, possibly empty, argument list. It represents the value returned by the built-in function.



BUILT-IN FUNCTIONS

If the built-in function has arguments associated with it, you must specify an argument list that has the number of arguments required by the built-in function. If the built-in function has no arguments, you may specify the empty argument list or omit the argument list.

Built-in functions include the commonly used arithmetic functions and other necessary or useful functions related to language facilities, such as functions for manipulating strings or converting data.

Classification of Built-In Functions

The built-in functions are classified as follows:

- Computational built-in functions
 - String handling
 - Arithmetic
 - Mathematical
 - Array handling
- Condition handling
- Storage control
- Input/output
- Miscellaneous

String Handling Built-In Functions

The string handling built-in functions simplify the processing of bit and character values. They are:

| | |
|-----------|-----------|
| BIT | SUBSTR |
| CHARACTER | TRANSLATE |
| COPY | UNSPEC |
| INDEX | VERIFY |
| LENGTH | |

Arithmetic Built-In Functions

The arithmetic built-in functions control the conversion of base, scale, and precision, and determine the properties of arithmetic values. The arithmetic built-in functions are:

| | |
|---------|-------|
| ABS | MAX |
| BINARY | MIN |
| DECIMAL | MOD |
| DIVIDE | ROUND |
| FIXED | SIGN |
| FLOAT | TRUNC |

Some of these functions derive the data type of their results from one or more arguments. When the base or scale of the arguments differ, they are converted to their common base and scale. If the scales differ, fixed-point is converted to floating-point. If the bases differ, decimal is converted to binary.

Except where otherwise stated, picture arguments are converted to fixed-point decimal.

To determine the target precision of an argument that is to be converted to an arithmetic type, refer to Figure 12-4 on page 12-27. If the argument is:

- Bit, use FIXED BINARY(31) as the source.
- Character, use FIXED DECIMAL(15,0) as the source.

For functions that give a binary fixed result or a floating-point result, the precision specified does not cause rounding or truncation of the result. It does, however, determine the storage requirements of the result (see Figure 5-1 on page 5-10).

Mathematical Built-In Functions

The mathematical built-in functions provide mathematical operations. They are:

| | |
|-------|-------|
| ACOS | LOG2 |
| ASIN | LOG10 |
| ATAN | SIN |
| ATAND | SIND |
| ATANH | SINH |
| COS | SQRT |
| COSD | TAN |
| COSH | TAND |
| EXP | TANH |
| LOG | |

These functions operate on floating-point values to produce a floating-point result. Any arithmetic argument that is not floating-point is converted (see “Data Conversion” on page 5-27).

The range of values for each argument of the mathematical built-in functions can be found listed with the function.

Array Handling Built-In Functions

The array handling built-in functions operate on array arguments and return a scalar value. The array may have inherited dimensions. The array handling built-in functions are:

| | |
|-----------|--------|
| DIMENSION | LBOUND |
| HBOUND | |

Condition Handling Built-In Functions

The condition handling built-in functions allow you to investigate the cause of a raised condition. Use of a condition handling built-in function is **in context** when within an on-unit or dynamic descendant of an on-unit whose activation sets the value of the function. This is described for each condition handling built-in function below. See also “Scope of Values of Condition Handling Built-In Functions” on page 10-4. The condition handling built-in functions are:

BUILT-IN SUBROUTINES

ONCODE ONKEY
ONFILE

Storage Control Built-In Functions

The storage control built-in functions allow you to identify the location of a variable, or create a null pointer value, or calculate the amount of storage (in bytes) allocated to a variable. The storage control built-in functions are:

ADDR STORAGE
NULL

Input/Output Built-In Functions

The input/output built-in functions allow you to determine the current state of a file. They are:

LINENO
SAMEKEY

Miscellaneous Built-In Functions

The built-in functions that do not fit into any of the foregoing classes are:

DATE PLISHUTDN
PLIRETV TIME

Built-In Subroutines

A built-in subroutine is a predefined routine that provides access to facilities of the operating system. It is called by a CALL statement (see "CALL Statement" on page 14-7).

You can explicitly declare a built-in name to have the BUILTIN attribute (see "BUILTIN Attribute" on page 12-36).

The built-in subroutines, and their uses, are:

| IBM Extension | |
|----------------------|---|
| PLIDUMP | Gives a symbolic dump of the variables of the currently running PL/I program. |
| PLIRETC | Allows a program to receive a PL/I program return code. |
| End of IBM Extension | |
| PLICOMMIT | Establishes a commitment boundary and processes commitment control functions. |

| | |
|--------------------|--|
| PLIIOFDB | Copies information from the system-defined I/O feedback area. |
| PLIOPNFDB | Copies the system defined open feedback area. |
| PLIRCVMMSG | Returns information about the original message that caused the calling of an on-unit. |
| PLIROLLBACK | Reestablishes a previous commitment boundary and processes commitment control functions. |

You can explicitly declare a built-in name to have the BUILTIN attribute (see "BUILTIN Attribute" on page 12-36).

Pseudovariabes

Pseudovariabes represent targets for assignments. A pseudovariabes can appear only on the left of the assignment symbol in an assignment statement.

The pseudovariabes are SUBSTR and UNSPEC.

Pseudovariabes cannot be nested. For example, the following statement is not valid:

```
UNSPEC(SUBSTR(A,2,1)) = '04'B4;
```

Aggregate Arguments

The built-in functions that can accept aggregate arguments are ADDR and STORAGE, and if the aggregate is an array, DIMENSION, HBOUND, and LBOUND.

Empty Argument Lists

Some built-in functions do not require arguments. You must declare these either explicitly with the BUILTIN attribute or contextually by including an empty argument list in the built-in function reference, as in ONKEY(). The name cannot otherwise be recognized as a built-in function name.

You specify an empty argument list by following the function name with an open parenthesis followed immediately by a close parenthesis.

Descriptions of Built-In Functions, Subroutines and Pseudovariabes

Unless otherwise noted, the following items are built-in functions. Options enclosed within square brackets are optional.

ABS(x)

ABS returns the absolute value of *x*.

x

An arithmetic or picture expression.

The value returned by ABS is the absolute value of *x*. The result has the base, scale, and precision of *x*.

ACOS(x)

ACOS returns a floating-point value that represents the arc cosine, in radians, of x.

x

An arithmetic or picture expression, where $ABS(x) \leq 1$.

The result is in the range:

$$0 \leq ACOS(x) \leq \pi$$

and has the base and precision of x.

ADDR(x)

ADDR returns a pointer value that identifies the location of the variable x in storage.

x

A reference to a variable of any data type, data organization, alignment, and storage class. In a reference to an array with inherited dimensions, subscripts for the inherited dimensions must be specified.

If x is a reference to:

- An aggregate, the returned value identifies the first structure field or array element.
- A component of an aggregate, the returned value takes into account subscripting and structure qualification.
- A based level-1 variable, the result is the value of the pointer that explicitly qualifies x (if it appears) or that is associated with x in its declaration.
- A parameter for which a dummy argument has been created, the returned value identifies the dummy argument.
- A varying-length character string, the returned value points to the halfword prefix at the beginning of the string.

ASIN(x)

ASIN returns a floating-point value that represents the arc sine, in radians, of x.

x

An arithmetic or picture expression, where $ABS(x) \leq 1$.

The result is in the range:

$$-\pi/2 \leq ASIN(x) \leq \pi/2$$

and has the base and precision of x.

ATAN(x[,y])

ATAN returns a floating-point value that represents the arc tangent, in radians, of x or of the ratio x/y .

x,y

Arithmetic or picture expressions.

If you omit y , the result has the base and precision of x and is in the range:

$$-\pi/2 < \text{ATAN}(x) < \pi/2$$

If you specify x and y , they must not both be zero. The result for all other values of x and y has the common base of the arguments. x and y are converted to the base and scale of the result. The result precision is the larger of those of the converted arguments. The value is given by:

| | |
|-------------------------|-------------------------|
| ATAN(x,y) | for $y>0$ |
| $\pi/2$ | for $y=0$ and $x>0$ |
| $-\pi/2$ | for $y=0$ and $x<0$ |
| $\pi+\text{ATAN}(x,y)$ | for $y<0$ and $x\geq 0$ |
| $-\pi+\text{ATAN}(x,y)$ | for $y<0$ and $x<0$ |

Therefore ATAN(x,y) returns the value, in the range

$$-\pi < \text{ATAN}(x,y) \leq \pi$$

of the point with rectangular coordinates y and x .

ATAND(x[,y])

ATAND returns a floating-point value that represents the arc tangent, in degrees, of x or of the ratio x/y .

x,y

Arithmetic or picture expressions.

If you specify x alone, the result has the base and precision of x and is in the range:

$$-90 < \text{ATAND}(x) < 90$$

If you specify x and y , the value of the result is given by:

$$(180/\pi) * \text{ATAN}(x,y)$$

See "ATAN($x[,y]$)" for the requirements of the arguments and the attributes of the result.

ATANH(x)

ATANH returns a floating-point value that has the base and precision of x , and represents the hyperbolic arc tangent of x , in radians.

x

An arithmetic or picture expression.

The result has a value given by:

$$\text{LOG}((1+x)/(1-x))/2$$

BINARY(x[,p[,0]])

Abbreviation: **BIN**

BINARY converts *x* to a binary value with a precision specified by *p*.

x

An arithmetic or string expression.

p

An integer constant that specifies the number of digits of the result. It must not exceed the implementation limit (31 for fixed-point or 53 for floating-point).

You must specify *p* if *x* is fixed-point decimal or pictured with a nonzero scale factor.

If you omit *p*, the precision of the result is the converted precision of *x* (see Figure 12-4 on page 12-27).

If *x* is arithmetic, the scale of the result is that of *x*. If *x* is a string, the scale of the result is fixed-point binary.

BIT(x[,y])

BIT converts *x* to a bit string with a length specified by *y*.

x

An arithmetic or string expression. When a character string expression is used, it must contain only ones and zeros.

y

An integer expression with a non-negative value. If you omit *y*, **BIT** determines the length according to the rules for type conversion (see "Data Conversion" on page 5-27). If *y*=0, the result is a null bit string.

CHARACTER(x[,y])

Abbreviation: **CHAR**

CHARACTER converts *x* to a character string with a length specified by *y*.

x

An arithmetic or string expression.

y

An integer expression with a non-negative value. If you omit *y*, **CHARACTER** determines the length according to the rules for type conversion (see "Data Conversion" on page 5-27). If *y*=0, the result is a null character string.

COPY(x,y)

COPY returns a string consisting of *y* concatenated copies of the string *x*.

x

A string expression.

y
An integer expression with a non-negative value that specifies the number of repetitions.

If y is zero, the result is a null string.

COS(x)

COS returns a floating-point value that has the base and precision of x and represents the cosine of x, in radians.

x
An arithmetic or picture expression whose value is in radians.

COSD(x)

COSD returns a floating-point value that has the base and precision of x and represents the cosine of x, in degrees.

x
An arithmetic or picture expression whose value is in degrees.

COSH(x)

COSH returns a floating-point value that has the base and precision of x, and represents the hyperbolic cosine of x, in radians.

x
An arithmetic or picture expression.

The result is in the range:

$$+1 \leq \text{COSH}(x) \leq +\text{infinity}$$

DATE[()]

DATE returns a character string of length 6, in the form yymmdd, where:

yy the last two digits of the current year
mm the current month
dd the current day

DECIMAL(x[,p[,q]])

Abbreviation: DEC

DECIMAL converts x to a decimal value with a precision specified by p and q.

x
An arithmetic or string expression.

p
An integer constant that specifies the number of digits of the result. It must not exceed the implementation limit (15 for fixed-point or 16 for floating-point).

q
An integer constant that specifies the scale factor of the result. For a fixed-point result, if you specify *p* and omit *q*, a scale factor of zero is assumed. For a floating-point result, *q* must be omitted.

If you omit both *p* and *q*, the precision of the result is the converted precision of *x* (see Figure 12-4 on page 12-26).

If *x* is arithmetic, the result has the scale of *x*. If *x* is a string, the scale of the result is decimal fixed-point.

DIMENSION(x,y)

Abbreviation: **DIM**

DIMENSION returns a fixed-point binary value of precision (15,0) that specifies the extent of dimension *y* of array *x*.

x
An array reference. The array can have inherited dimensions.

y
An integer constant that specifies a particular dimension of *x*.

x must not have fewer than *y* dimensions. *y* must be greater than or equal to 1.

As the lower bound of an array dimension is always 1, the extent of the dimension is the same as its upper bound. Therefore **DIM(x,y)** is equal to **HBOUND(x,y)**.

DIVIDE(x,y,p[,q])

DIVIDE returns the result of *x* divided by *y*. The precision of the result is specified by *p* and *q*.

x
An arithmetic or picture expression that represents the dividend.

y
An arithmetic or picture expression that represents the divisor. If *y* = 0, the zerodivide condition is raised.

p
An integer constant that specifies the number of digits to be maintained throughout the operation. It must not exceed the maximum number of digits allowed for the base and scale of the result.

q
An integer constant that specifies the scale factor of the result, which must be fixed-point decimal. If you omit *q*, the **DIVIDE** built-in function assumes a scale factor of zero. *q* must be zero, if specified, for a fixed-point binary result or omitted for a floating-point result.

If either *x* or *y* is fixed-point binary, the other expression must not be fixed-point decimal with a nonzero scale factor.

The base and scale of the result are the common base and scale of x and y .

EXP(x)

EXP returns a floating-point value that represents the base e of the natural logarithm system raised to the power of x .

x

An arithmetic or picture expression that is the power to which the base is raised.

The result has the base and precision of x .

FIXED($x,p[,q]$)

FIXED converts x to a fixed-point value with a precision specified by p and q .

x

An arithmetic or string expression.

p

An integer constant that specifies the number of digits of the result. It must not exceed the implementation limit (15 for decimal or 31 for binary).

q

An integer constant that specifies the scale factor of the result. If you omit q , the FIXED built-in function assumes a scale factor of zero.

For a binary result, q must be zero, if specified.

If x is a string, the base of the result is binary for a bit string and decimal for a character string. Otherwise, the result has the base of x .

FLOAT(x,p)

FLOAT converts x to a floating-point value with a precision specified by p .

x

An arithmetic or string expression.

p

An integer constant that specifies the number of digits of the result. It must not exceed the implementation limit (16 for decimal or 53 for binary).

If x is a string, the base of the result is binary for a bit string and decimal for a character string. Otherwise the result has the base of x .

HBOUND(x,y)

HBOUND returns a fixed-point binary value of precision (15) that specifies the upper bound of dimension y of array x .

x

An array reference. The array can have inherited dimensions.

y

An integer constant that specifies a particular dimension of x .

x must not have fewer than y dimensions. y must be greater than or equal to 1.

INDEX(x,y)

INDEX returns a binary fixed-point value of precision (15), which indicates the starting position within string x of a substring identical to string y.

x
A string expression that specifies the string searched.

y
A string expression that specifies the string searched for.

If y does not occur in x, or if either x or y has zero length, INDEX returns the value zero.

If y occurs more than once in x, INDEX returns the starting position of the left-most occurrence.

Both arguments must be bit strings or both must be character strings.

LBOUND(x,y)

LBOUND returns a fixed-point binary value of precision (15) that specifies the lower bound of dimension y of the array x. This value is always 1.

x
An array reference. The array can have inherited dimensions.

y
An integer constant that specifies a particular dimension of x.

x must not have fewer than y dimensions. y must be greater than or equal to 1.

LENGTH(x)

LENGTH returns a fixed-point binary value of precision (15) that specifies the length of string x.

x
A bit expression or a character expression.

LINENO(x)

LINENO returns a fixed-point binary value with precision (15) that specifies the current line number of file x.

x
A file constant. The file must be open and have the PRINT attribute.

LOG(x)

LOG returns a floating-point value that has the base and precision of x and represents the natural logarithm (that is, the logarithm to the base e) of x.

x
An arithmetic or picture expression greater than zero.

LOG2(x)

LOG2 returns a floating-point value that has the base and precision of x and represents the binary logarithm (that is, the logarithm to the base 2) of x.

x

An arithmetic or picture expression greater than zero.

LOG10(x)

LOG10 returns a floating-point value that has the base and precision of x and represents the common logarithm (that is, the logarithm to the base 10) of x.

x

An arithmetic or picture expression greater than zero.

MAX(x1,x2)

MAX returns the greater of x1 and x2

x1,x2

Arithmetic or picture expressions.

If either x1 or x2 is fixed-point binary, and the other is fixed-point decimal or picture, the other must have a scale factor of zero.

The result has the common base and scale of x1 and x2. The arguments are converted to the base and scale of the result.

If the converted arguments are fixed-point with precisions:

$(p1, q1), (p2, q2)$

the precision of the result is given by:

$p = \text{MIN}(n, \text{MAX}(p1 - q1, p2 - q2) + \text{MAX}(q1, q2))$

$q = \text{MAX}(q1, q2)$

where n is the maximum number of digits allowed (decimal 15, binary 31). For binary, $q1 = q2 = 0$.

If the converted arguments are floating-point with precisions:

$p1, p2$

the precision of the result is given by:

$p = \text{MAX}(p1, p2)$

MIN(x1,x2)

MIN returns the smaller of x1 and x2.

x1,x2

Arithmetic or picture expressions.

If either x_1 or x_2 is fixed-point binary, and the other is fixed-point decimal or picture, the other must have a scale factor of zero.

The result has the common base and scale of x_1 and x_2 . The arguments are converted to the base and scale of the result.

The precision of the result is the same as for the MAX built-in function.

MOD(x,y)

MOD returns the remainder obtained by dividing x by y , with the sign of y . If $y=0$, MOD returns x .

x,y
Arithmetic or picture expressions.

If either x_1 or x_2 is fixed-point binary, and the other is fixed-point decimal or picture, the other must have a scale factor of zero.

For $y \neq 0$, the result of MOD is $x-y*\text{floor}(x/y)$, where $\text{floor}(x/y)$ is the largest integer less than or equal to x/y .

The result has the common base and scale of x and y . The arguments are converted to the base and scale of the result.

If the result is floating-point, the result precision is the greater of those of x and y . If the result is fixed-point, the result precision is given by:

$$p = \text{MIN}(n, p_2 - q_2 + \text{MAX}(q_1, q_2))$$
$$q = \text{MAX}(q_1, q_2)$$

where (p_1, q_1) is the precision of x and (p_2, q_2) that of y after any necessary conversion, and n is the maximum number of digits allowed (15 for decimal or 31 for binary). For binary, $q_1 = q_2 = 0$.

NULL[()]

NULL returns the null pointer value.

The null pointer value cannot identify any variable.

ONCODE[()]

ONCODE returns a fixed-point binary value of precision (15), which is the current condition code.

Whenever an on-unit for a condition is entered, the condition code is set to a value that describes the situation that raised the condition. If ONCODE is used out of context, it returns zero.

Conditions and condition codes are described in Appendix D, "Conditions and Condition Codes." Condition handling is described in Chapter 10, "Condition Handling Statements."

ONFILE[()]

ONFILE returns a character string, which is the name of the file for which an input/output or conversion condition has been raised.

The value of ONFILE is set when an on-unit is entered for any of the following conditions:

- An input/output condition
- The conversion condition raised during an input/output operation
- The ERROR condition raised as the implicit action for such a condition.

If ONFILE is used out of context, it returns the null character value.

ONKEY[()]

ONKEY returns a character value, which is the key of the record for which an input/output condition has been raised. The value of ONKEY is set whenever an on-unit is entered for an input/output condition (except ENDFILE) caused by an input/output statement that contained the KEY or KEYFROM(expression), or for the ERROR condition raised as implicit action for such a condition. If the KEY or KEYFROM(expression) is not specified, or if KEYFROM(*) is specified, ONKEY returns a null character value. The variable returned may contain numeric and character fields. For CONSECUTIVE files, the BIN(31) value is converted to a character value and returned. For files which contain multiple key fields, each field is concatenated with the next field. Numeric fields appear in internal form unaligned.

The result is determined according to the following rules:

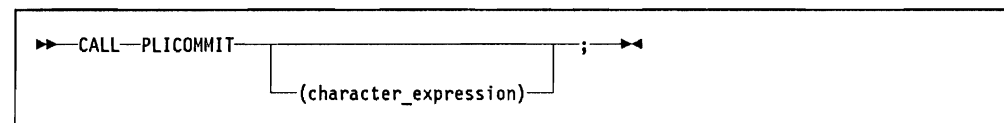
- The result is the key of the record that was processed by the input/output statement that caused the error.
- For the READ statement, the value in the KEY option is returned. If the value for NBRKEYFLDS in the OPTIONS option is less than the maximum for the record, the full KEY value may not be returned.
- For the WRITE statement, the value in the KEYFROM option is returned.
- For the REWRITE or DELETE statement with the KEY option specified, the value in the KEY option is returned.

Note: If the duplicate key condition is raised by the REWRITE statement, the key value in the key option, and the key value imbedded in the record may not be the same, therefore the ONKEY built-in function may not return the value of the duplicate key.

If ONKEY is used out of context, it returns the null character value.

PLICOMMIT Built-In Subroutine

The PLICOMMIT built-in subroutine processes commitment control functions.



character_expression

A character expression that can be converted to a non-varying character variable. A character string of zero length is equivalent to not specifying the argument.

For more information about how to use PLICOMMIT, see “Commitment Control” on page 8-58.

IBM Extension

PLIDUMP Built-In Subroutine

The PLIDUMP built-in subroutine produces a symbolic dump of the variables of the currently running PL/I program.

```
▶—CALL—PLIDUMP—( 'options_list' , user_identification )—;—▶
```

The program variables that are dumped depend on the options you specify when you call PLIDUMP. The dump also contains:

- A list of any ONCODE, ONFILE, or ONKEY data which is relevant
- The date and time of the dump
- The statement number from which the dump was called.

You can also request this dump whenever the program encounters a system error that is not handled by OS/400 or by the program.

For a complete description of how to use PLIDUMP, see “Using PLIDUMP” on page 3-14.

End of IBM Extension

PLIIOFDB Built-In Subroutine

The PLIIOFDB built-in subroutine copies the system defined I/O feedback area.

```
▶—CALL—PLIIOFDB(file_constant,reference)—;—▶
```

file_constant

The declaration file name.

reference

Either a non-varying scalar or a connected structure reference that will contain the feedback data. If a structure reference is used, all binary fields in the structure should be declared UNALIGNED. If the length of the reference supplied

is less than the feedback data available, the data is truncated. If the length is greater than the feedback data available, the remaining portion of the reference is not modified.

The system defined I/O feedback area contains information about the last I/O operation processed, including:

- The last record format read
- The major and minor return codes from communications and display processing
- The last data base record key read.

The system defined I/O feedback area consists of a common area and a device-specific area.

The structure of these areas is shown in the *Programming: Control Language Programmer's Guide*.

PLIOPNFDB Built-In Subroutine

The PLIOPNFDB built-in subroutine copies the system defined open feedback area.

```
▶▶CALL—PLIOPNFDB(file_constant,reference)—;▶▶
```

file_constant

The declaration file name.

reference

Either a non-varying scalar or a connected structure reference that will contain the feedback data. If a structure reference is used, all binary fields in the structure should be declared UNALIGNED. If the length of the reference supplied is less than the feedback data available, the data is truncated. If the length is greater than the feedback data available, the remaining portion of the reference is not modified.

The first time a file is opened, system feedback information about the file is copied into the open feedback area. The content of the open feedback area does not change as long as the file status does not change.

The open feedback area contains information such as:

- The name of the library, file, and member opened
- The type of file opened
- For data base files:
 - Duplicate keys allowed
 - Duplicate key order
 - Commitment control information.

The structure of the system defined open feedback area is shown in the *Programming: Control Language Programmer's Guide*.

PLIRCVMSG Built-In Subroutine

PLIRCVMSG returns the message identifier, message data record and message reference key for the last escape or notify message that resulted in the calling of an on-unit. PLIRCVMSG is intended for use in an on-unit to determine the source of the error. This may be useful because multiple system messages may map to one PL/I condition code.

```
▶▶CALL PLIRCVMSG(message_id,message_data,message_reference_key);▶▶
```

message_id

A character variable that will contain the system message identifier of the last escape or notify message. This should be seven characters long.

message_data

A character variable that will contain the message data record. The message data record contains the substitution values (in a single character string) that were used in the text of the received message. This data varies with each message. If the length of the variable supplied is less than the message data available, the data will be truncated. If the length is greater than the message data available, the remaining portion of the variable will not be modified.

message_reference_key

A character variable that will contain the message reference key that identifies the message. This should be four characters long.

PLIRCVMSG will return the OS/400 escape or notify message that called the on-unit. STATUS messages cannot be received from a program message queue. If a STATUS message is the cause of the on-unit call, PLIRCVMSG will set the message identifier field to blanks. If the PL/I compiler is originating the condition the message returned will be a PL/I defined message.

If the original message was CPF9999, PLIRCVMSG will return the message identifier, message data record, and message reference key of the escape message that caused CPF9999.

If PLIRCVMSG is not used within an on-unit, it will return blanks in the variables. If PLIRCVMSG is called multiple times within an on-unit, it will return the same values as the previous use.

IBM Extension

PLIRETC Built-In Subroutine

PLIRETC passes a return code from a called PL/I program to the program that called the PL/I program. The return code is passed in the return code field in the Work Control Block (WCB). This allows the passing of the return code either for examination by a program (BASIC, CL, PL/I, RPG) in a subsequent job step or it may be used to indicate conditions that were encountered while running.


```
▶▶—CALL—PLIRETC(expression)—;—▶▶
```

expression

An expression which is converted to FIXED BINARY (15,0).

The return code generated by a PL/I program consists of a BIN(15) field and is passed to PL/I program management (PL/I) with a call to PLIRETC. At the beginning of a run-unit the return code field in the WCB is set to zero. Thereafter PL/I will only store a value in the WCB when a run-unit ends abnormally, or when the program stores a specified return code by calling PLIRETC.

If PL/I ends abnormally the return code indicates the way in which the program ended, unless an error is detected which prevents the PL/I program management routines from operating correctly. Therefore if PL/I ends normally any value set using PLIRETC or which was set by a called lower level procedure will remain in the return code in the WCB.

When a PL/I program calls PLIRETC, the argument (return code value) can be either a constant or a variable with the attributes FIXED BINARY (15,0).

The following table shows the values and meanings of the return codes generated by the PL/I program management routines. Any values greater than 0, other than those in the following list was set either by PLIRETC or by a called procedure.

| Return Code | Meaning |
|--------------------|--|
| 0 | Normal ending. |
| 2 | STOP statement, or a call to PLIDUMP with the S option. |
| 3 | ERROR condition raised and run-unit ended with return from ERROR on-unit or no active ERROR on-unit was found. |
| 4 | PL/I program management routines detected an error which did not allow the ERROR on-unit to be called. |

End of IBM Extension

PLIRETV[()]

PLIRETV returns a fixed-point binary value of precision (15,0) that is the PL/I return code.

The value of the return code is one of the following:

- The last value specified by a CALL PLIRETC statement.
- The value returned by an external procedure.
- Zero.

An empty argument list may be used to declare the function as BUILTIN.

PLIROLLBACK Built-In Subroutine

The PLIROLLBACK built-in subroutine processes the rollback function by reestablishing a previous commitment boundary.

```
▶▶—CALL—PLIROLLBACK—;—▶▶
```

For more information on PLIROLLBACK, see "Commitment Control" on page 8-58.

PLISHUTDN[()]

PLISHUTDN returns a BIN(15) FIXED variable which indicates the system status request to end the program.

A value of 127 indicates that a controlled cancel is pending. A value of 0 indicates that no cancel is pending.

An empty argument list may be used to declare the function as BUILTIN.

If the CL command ENDJOB is issued with *CNTRLD specified as the OPTION parameter, the job does not end until the length of time specified in the DELAY parameter has passed.

ROUND(x,y)

The value of x is rounded at the digit position specified by y.

x

A fixed-point decimal or picture expression with a nonzero scale factor.

y

A non-negative integer constant that specifies the digit position at which the value is rounded; that is, at digit y to the right of the decimal point.

The precision of the fixed-point result is given by:

$$p = \text{MAX}(1, \text{MIN}(p-q+1+y, 15))$$

$$q = y$$

where (p,q) is the precision of x.

The result has the base and scale of x.

SAMEKEY(x)

The SAMEKEY built-in function is included for compatibility with other implementations of PL/I. AS/400 allows you to find a common file key by specifying POSITION (NXTEQL) on the OPTIONS option of an input/output statement (see "POSITION Parameter" on page 7-17).

SAMEKEY returns a bit value of length 1 that indicates if a record that has been accessed is followed by another with the same key.

x

A file constant. The file must have the RECORD attribute.

Upon successful completion of an input/output operation on file x or immediately before raising the ERROR condition, the value accessed by SAMEKEY is set to '1'B if the record processed is followed by another record with the same key. Otherwise, the value is set to '0'B.

The value accessed by SAMEKEY is also set to '0'B if:

- The file does not have the attributes INDEXED, RECORD, and INPUT or UPDATE.
- An input/output operation that raised a condition other than ERROR also changed or lost file positioning.
- The file is not open.
- A current file position does not exist.
- A current file position is the last record in the file in the direction of data transfer.

Note: a return value of '1'B only guarantees that the next record has the same key at the time the value was returned. It does not guarantee that the key will be the same when you access it, or that you will be able to access the record even if the key is the same.

End of IBM Extension

SIGN(x)

SIGN returns a fixed-point binary value of precision (15) that indicates if x is positive, zero, or negative.

x

An arithmetic or picture expression.

The returned value is given by:

Value of x *Value Returned*

| | |
|---------|----|
| $x > 0$ | +1 |
| $x = 0$ | 0 |
| $x < 0$ | -1 |

SIN(x)

SIN returns a floating-point value that has the base and precision of x and represents the sine of x , in radians.

x
An arithmetic or character expression with a value measured in radians.

SIND(x)

SIND returns a floating-point value that has the base and precision of x and represents the sine of x in degrees.

x
An arithmetic or character expression with a value measured in degrees.

SINH(x)

SINH returns a floating-point value that has the base and precision of x , and represents the hyperbolic sine of x , in radians.

x
An arithmetic or character expression with a value measured in radians.

SQRT(x)

SQRT returns a floating-point value that has the base and precision of x and represents the positive square root of x .

x
An arithmetic or character expression. The value of x must not be less than zero.

IBM Extension

STORAGE(x)

Abbreviation: **STG**

STORAGE returns a fixed-point binary value of precision (31,0) giving the implementation-defined storage, in bytes, allocated to a variable x .

x
A connected variable of any data type, data organization, alignment, and storage class.

If *x* is a varying-length string, the value returned by `STORAGE(x)` is the sum of the length-prefix of the string and the number of bytes in the maximum length of the string. If *x* is an aggregate containing varying-length strings, the value returned by `STORAGE(x)` is the sum of the length prefixes of the strings and the number of bytes in the maximum lengths of the strings.

End of IBM Extension

SUBSTR(x,y[,z])

The `SUBSTR` built-in function returns the substring, specified by *y* and *z*, of string *x*.

x

A bit or character expression.

y

An integer expression that indicates the starting position of the substring in *x*.

z

An integer expression that specifies the length (number of bits or characters) of the substring in *x*. If *z* is a negative constant then a compile time error will occur. If *z* is the zero constant or is a variable and is assigned a negative value, then a null string is returned. If *z* is omitted, the substring extends to the end of *x*.

Assuming that *x*, *y*, and *z* represent their corresponding values, the following relationship is true:

$$1 \leq y \leq \text{length}(x) + 1$$

$$0 \leq z \leq \text{length}(x) - y + 1$$

Therefore, the starting position *y* may be one position after the end of the string if the length of the substring is zero.

SUBSTR(x,y[,z]) Pseudovvariable

The `SUBSTR` pseudovvariable identifies the substring, specified by *y* and *z*, of the string variable *x*. You can use the `SUBSTR` pseudovvariable as the target in an assignment statement to assign a string value to the substring.

x

A reference to a string variable.

y

An integer expression that indicates the starting position of the substring in *x*.

z

An integer expression that specifies the length of the substring in *x*. If *x* is a negative constant then a compile time error will occur. If *z* is the zero constant or is a variable and is assigned a negative value, then a null string is returned. If *z* is omitted, the substring extends to the end of *x*.

The same relationships must hold for the arguments as are given for the `SUBSTR` built-in function.

TAN(x)

TAN returns a floating-point value that has the base and precision of *x* and represents the tangent of *x*, in radians.

x

An arithmetic or picture expression whose value is in radians.

TAND(x)

TAND returns a floating-point value that has the base and precision of *x* and represents the tangent of *x*, in degrees.

x

An arithmetic or picture expression whose value is in degrees.

TANH(x)

TANH returns a floating-point value that has the base and precision of *x*, and represents the hyperbolic tangent of *x*, in radians.

x

An arithmetic or picture expression whose value is in radians.

The range of the result is:

$$-1 \leq \text{TANH}(x) \leq +1$$

TIME[()]

TIME returns a fixed-length character string of length 9, hhmmss_{ttt}, where:

| | |
|-----|-------------------------|
| hh | the current hour |
| mm | the current minute |
| ss | the current second |
| ttt | the current millisecond |

TRANSLATE(x,y[,z])

TRANSLATE returns a character string of the same length as *x*; the characters are translated according to the correspondence described by *y* and *z*.

x

A character expression that specifies the string translated.

y

A character expression that specifies the characters into which to translate.

z

A character expression that specifies the characters from which to translate. If *z* is omitted, a string of 256 characters is assumed. The string contains the EBCDIC characters in ascending collating sequence (hexadecimal 00 through FF).

y is padded with blanks or truncated on the right to match the length of *z*.

TRANSLATE translates each character of x as follows:

A character that occurs in z is translated into the character that occurs at the corresponding position in y. Any character that does not occur in z is left unchanged. (If a character occurs in z more than once, the leftmost occurrence is considered.)

For example:

```
DECLARE (W, X) CHAR (3);
X='ABC';
W = TRANSLATE (X, 'TAR', 'DAB');
```

In the above example, W is equal to 'ARC'.

TRUNC(x)

TRUNC returns an integer that is the truncated value of x. If x is greater than or equal to zero, the result is the largest integer less than or equal to x. If x is negative, the result is the smallest integer greater than or equal to x.

x

An arithmetic or picture expression.

The base, scale, and precision of the result match those of x, except when x is fixed-point decimal with precision (p,q). The precision of the result is given by:

```
p = MIN(15,MAX(p-q+1,1))
q = 0
```

UNSPEC(x)

The UNSPEC built-in function returns a bit value that is the internal coded form of the value represented by x.

x

A reference to a scalar variable.

The length of the returned bit value depends on the data type of x. If x is of type BIT(n), the length of the returned value is n. For any other data type, the length of the returned value is 8*n bits, where n is the length of x in bytes, as defined in Figure 5-1 on page 5-10.

UNSPEC(x) Pseudovisible

The UNSPEC pseudovisible identifies a target that is variable x considered as a bit variable.

x

A reference to a scalar variable.

The length of the bit variable is the same as for the UNSPEC built-in function. If UNSPEC(x) is used as the target in an assignment statement, the source value is converted to a bit value. The bit value is considered the internal representation of the value of x. It is assigned directly to x, without conversion, and is padded or truncated to the length of UNSPEC(x) according to the rules for bit assignment.

VERIFY(x,y)

VERIFY returns a fixed-point binary value of precision (15) that indicates the position in *x* of the leftmost character that is not in *y*. If all the characters in *x* appear in *y*, VERIFY returns a value of zero. If *x* is the null string, VERIFY returns a value of zero. If *x* is not the null string and *y* is the null string, VERIFY returns a value of 1.

x

A character expression that specifies the string scanned.

y

A character expression that specifies the characters searched for.

Appendix A. Compiler Service Information

This appendix is provided for PL/I service personnel to use when investigating PL/I compiler problems and provides the following information:

- Compiler overview
- Compiler debugging options
- IRP layout
- Quantitative limits of the compiler.

PL/I programmers can also use this information to investigate PL/I compiler problems on their own before, or instead of, calling for service.

Compiler Overview

This section provides the following compiler information:

- How the compiler works
- Compiler description
- Description of major compiler data area
- Organization of compiler error messages.

This section provides an internal view of the compiler. If you need an external view to investigate a PL/I problem, see Chapter 2, "Creating, Compiling, and Running Your PL/I Program," which describes entering a PL/I program into the system, compiling your program and using the listings that the compiler produces.

Figure A-1 on page A-2 summarizes how a PL/I source program is compiled into a program object.

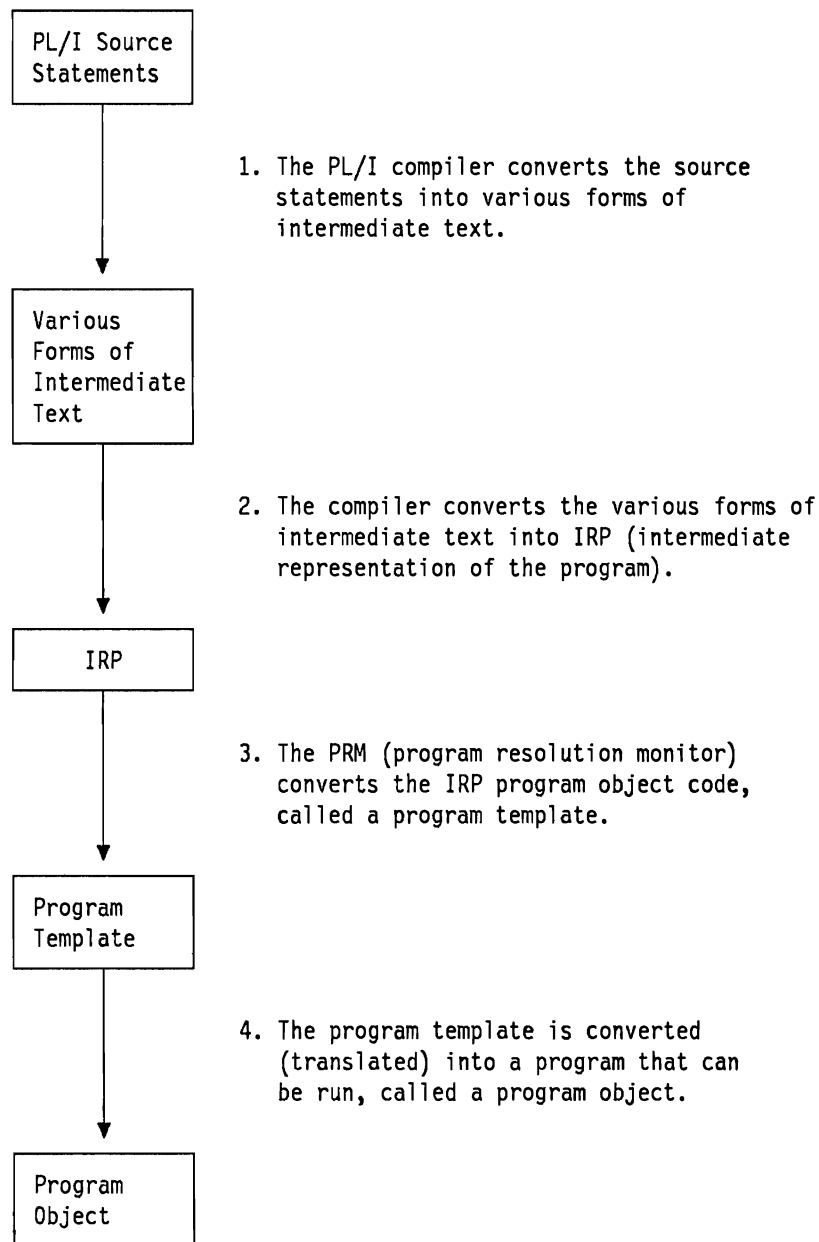


Figure A-1. Overview of the Compilation of a PL/I Program.

The various forms of intermediate text are representations of PL/I source statements that are created by the compiler and exist only during compilation.

You can use the SERVICE parameter of the CRTPLIPGM command to list the various forms of intermediate text. See "Compiler Debugging Options" on page A-10 for explanations of this parameter and examples of intermediate text.

When you compile the program, you can use the *LIST value for the GENOPT parameter on the CRTPLIPGM command to list the Intermediate Representation of the Program (IRP). See “Compiler Debugging Options” on page A-10 for an explanation of this parameter and examples of an IRP listing.

When you compile the program, you can use the *DUMP value for the GENOPT parameter on the CRTPLIPGM command to list the program template. See “Compiler Debugging Options” on page A-10 for an explanation of this parameter and an example of a program template listing.

Compiler Organization

The compiler consists of a number of modules, phases, and segments. Their relationships are shown in Figure A-2.

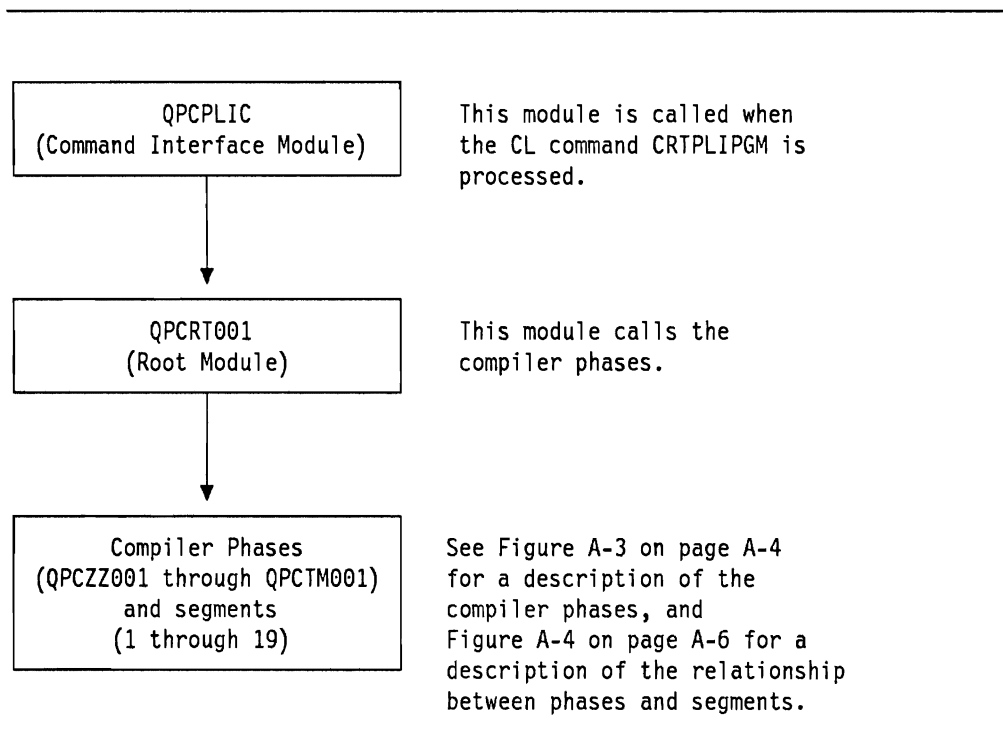


Figure A-2. Overview of the PL/I Compiler

The phases process the source statements and break them down into various forms of intermediate text. The various forms of intermediate text are stored in segments.

Compiler Phases

The compiler phases are described in Figure A-3 in the order that they are called sequentially under the control of the root module. Each name consists of the following parts:

1. The prefix QPC
2. Two characters that identify the phase
3. The suffix 001

COMPILER OVERVIEW

| Phase | Phase Description |
|----------|--|
| QPCZZ001 | The service phase. Allows compiler debugging to be done. This phase is called only if SERVICE (*YES) is specified on the CRTPLIPGM command. |
| QPCLP001 | The low level parse phase. Parses program source statements into tokenized text. |
| QPCHP001 | The high level parse phase. Parses tokenized text into Polish text. |
| QPCSV001 | The sort symbol vector phase. Sorts parsed symbols by name, block number, and statement number. |
| QPCDB001 | The dictionary build, name resolution, and cross-reference phase. Builds the dictionary, processes name resolution, and prepares the cross-reference list. |
| QPCDX001 | The first de-nesting phase. Processes Polish text, and completes the dictionary build. |
| QPCDY001 | The second de-nesting phase. Tests both Polish text and cross references, and produces a cross-reference listing. |
| QPCEA001 | The expression analysis phase. Analyzes expressions for validity. |
| QPCGC001 | The global check phase. Verifies that expressions and references are contextually correct. |
| QPCAG001 | The aggregate processing phase. Maps aggregates and adjustable strings, and prints the aggregate table. |
| QPCAD001 | The addressing expansion phase. Processes addressing expansion functions. |
| QPCDI001 | The DO and IF statement processing phase. Replaces DO, IF and END statements of do-groups by n-address text. |
| QPCST001 | The stream I/O processing phase. Processes GET and PUT statements. |
| QPCRC001 | The record I/O processing phase. Processes file control blocks and file constants. |
| QPCFL001 | The flow of control processing phase. Generates prolog code for procedures, blocks, and ON statements. |
| QPCTR001 | The text transformation phase. Generates FIT (Final Intermediate Text). |
| QPCSK001 | The skeleton identification phase. Translates FIT into EFIT (Extended Final Intermediate Text). |

Figure A-3 (Part 1 of 2). Compiler Phase Descriptions

| Phase | Phase Description |
|----------|---|
| QPCSA001 | The storage allocation phase. Determines the layout of run-time storage and produces IRP text for declarations. |
| QPCDC001 | The define constant section phase. Converts constants to internal format and produces IRP text for constant declarations. |
| QPCCG001 | The code generation phase. Produces IRP text for processable statements. |
| QPCFA001 | The final assembly phase. Calls the PRM (program resolution monitor) and produces the program object. |
| QPCTM001 | The end phase. Provides the diagnostic output (messages) and ends the program compilation. |

Figure A-3 (Part 2 of 2). Compiler Phase Descriptions

Intermediate Text

The compiler phases break down the source statements into various forms of intermediate text. Some types of intermediate text are:

- Polish text
- N-Address text
- Constant strings
- Symbol vectors

The text is stored in segments (see "Compiler Segments" below) by the phases. It may be formatted and listed by using the SERVICE parameter of the CRTPLIPGM command (see "Compiler Debugging Options" on page A-10) and the IBM-supplied formatters (see "Formatters and Intermediate Text" on page A-9).

Compiler Segments

The various forms of intermediate text are stored in segments. There are 19 segments which are implemented as MI (machine interface) space objects. Each segment may have different contents at different times in the compilation. The relationship between phases and the contents of segments is shown in Figure A-4 on page A-6. The figure appears in two parts. It breaks at the DI phase: for clarity, this phase is repeated where the second part begins.

COMPILER OVERVIEW

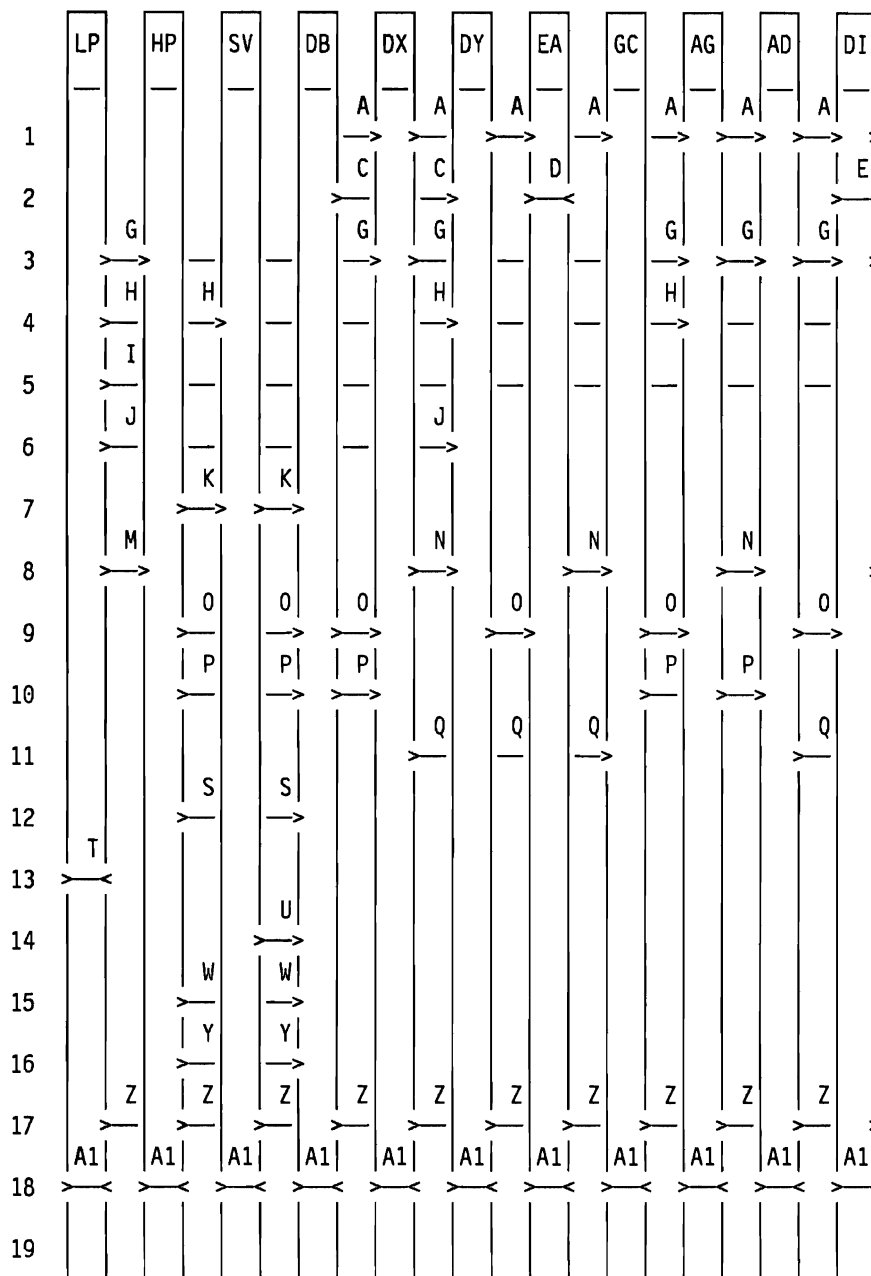


Figure A-4 (Part 1 of 3). Relationships between Phases and Segments

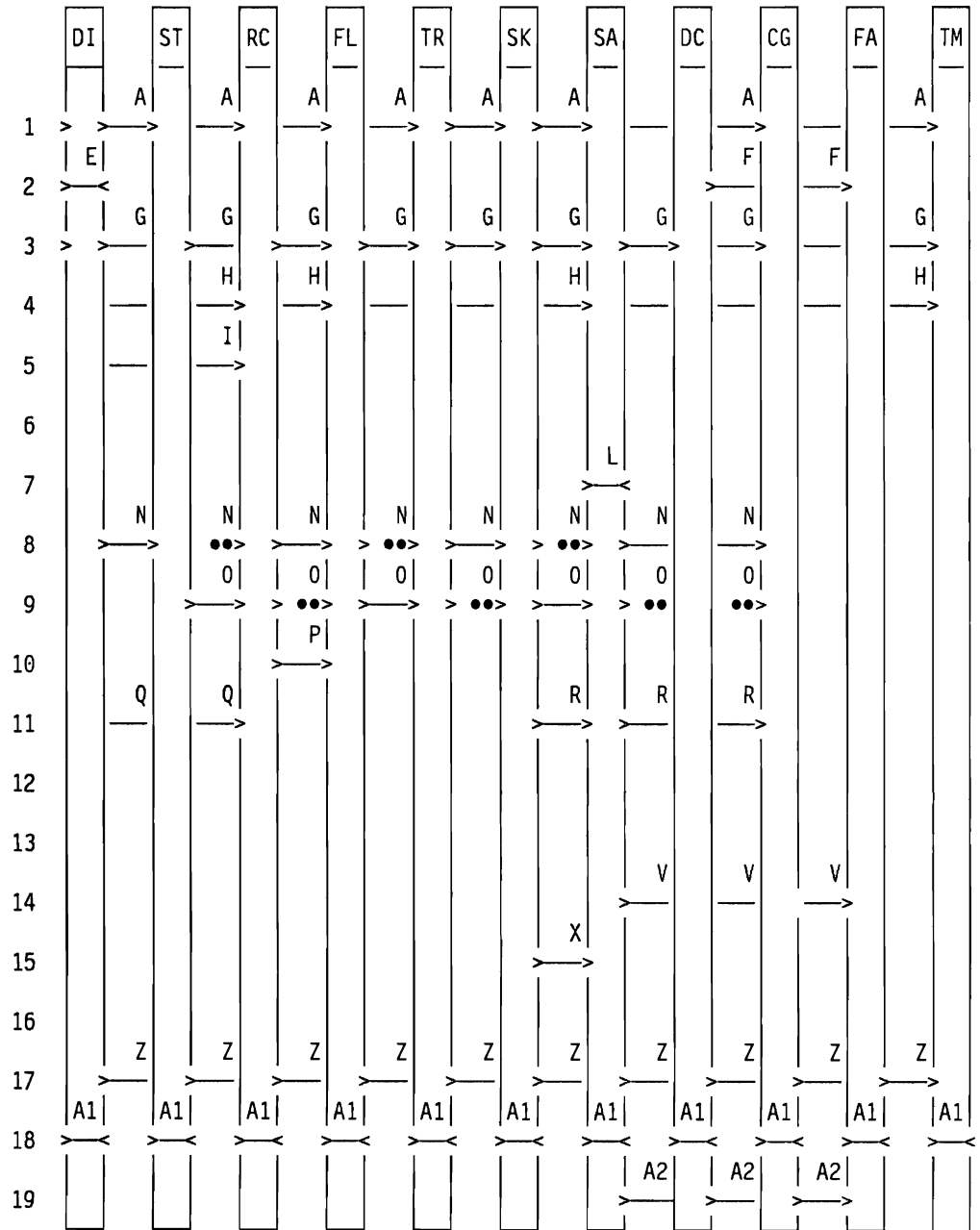


Figure A-4 (Part 2 of 3). Relationships between Phases and Segments

COMPILER OVERVIEW

Description of Symbols

| | |
|------|--------------------------------------|
| B | previous phase produces or changes B |
| → | following phase uses B as input |
| →••→ | alternate path |
| ← | internal workspace |

Figure A-4 (Part 3 of 3). Relationships between Phases and Segments

| Key | Segment Name |
|-----|----------------------------|
| A | Dictionary |
| C | Cross reference |
| D | EA stack |
| E | DI stack |
| F | DC associated space |
| G | Constant string |
| H | Name string |
| I | Back translate |
| J | Do nesting list |
| K | Symbol vector |
| L | Dictionary for temporaries |
| M | Tokenized text |
| N | Primary text |
| O | Primary text |
| P | Secondary text |
| Q | Secondary text |
| R | Block nearness array |
| S | Pre-dictionary elements |
| T | LP member stack space |
| U | Parameter vector |
| V | Where used |
| W | Qualification list |
| X | Branch point list |
| Y | Reorder vector |
| Z | Error messages |
| A1 | Work space |
| A2 | IRP |

Formatters and Intermediate Text

Formatters format and list the various forms of intermediate text so it can be used for debugging. Formatters are named like phases, except that the three digit number prefix may be any one from 001 through 019. The formatters and their descriptions are shown in Figure A-5.

| Formatter | Formatter Description |
|-----------|---|
| QPCPF008 | Formats and lists Polish-1 text, N-Address-1 text, FIT (final intermediate text), and EFIT (extended final intermediate text). |
| QPCPF009 | Formats and lists Polish-2 text and N-Address-2 text. |
| QPCDF003 | The dictionary formatter. Formats and lists information about each identifier in the program. |
| QPCTF001 | The tokenized text formatter. Formats and lists a condensed representation of the source program consisting of fixed-length tokens. |
| QPCCF001 | The constant string formatter. Formats and lists all the literals (arithmetic and string constants). |

Figure A-5. Compiler Segment Formatters

Error Message Organization

Error message numbers indicate the compiler phase from which they are called. Figure A-6 lists the range of the message numbers that are called from each phase.

| Phase | Message Range |
|----------|---------------|
| QPCRT001 | 0-299 |
| QPCLP001 | 1500-1799 |
| QPCHP001 | 1800-2399 |
| QPCSV001 | 2400-2699 |
| QPCDB001 | 2700-2999 |
| QPCDX001 | 3000-3299 |
| QPCDY001 | 3300-3599 |
| QPCEA001 | 3600-3899 |
| QPCGC001 | 3900-4199 |
| QPCAG001 | 4200-4499 |
| QPCAD001 | 4500-4799 |

Figure A-6 (Part 1 of 2). Error Messages Called by Compiler Phase

COMPILER DEBUGGING OPTIONS

| Phase | Message Range |
|----------|---------------|
| QPCDI001 | 4800-5099 |
| QPCST001 | 5100-5399 |
| QPCRC001 | 5400-5699 |
| QPCFL001 | 7500-7799 |
| QPCTR001 | 5700-5999 |
| QPCSK001 | 7800-8099 |
| QPCSA001 | 6000-6299 |
| QPCDC001 | 8100-8399 |
| QPCCG001 | 6300-6599 |
| QPCFA001 | 6900-7199 |
| QPCTM001 | 900-1199 |

Figure A-6 (Part 2 of 2). Error Messages Called by Compiler Phase

Compiler Debugging Options

The GENOPT and SERVICE parameters of the CL command CRTPLIPGM can be used to help debug PL/I problems. For information on the other parameters, see “Compiling Your Source Program Using the CRTPLIPGM Command” on page 2-5. For examples of debugging information that can be requested by these parameters, see “Examples of Using Compiler Debugging Options” below.

Examples of Using Compiler Debugging Options

Figure A-7 on page A-11 shows examples of debugging information that can be requested on the CL command CRTPLIPGM using the GENOPT keyword. The compiler listing in Figure A-7 on page A-11 was printed using a CRTPLIPGM command that specified debugging parameters as follows:

```
CRTPLIPGM QTEMP/LP1413 PLITST/PLISRC +  
  OPTION(*XREF *OPT *AGR *ATR) +  
  GENOPT(*LIST *XREF *PATCH *DUMP *ATR *DIAGNOSE)
```

The program compiled can be seen at Figure 8-3 on page 8-5. The parameters specified for the keyword OPTION are discussed at “Compiler Output” on page 2-19.

5728SS1 R01 M00 880715

GENERATED OUTPUT

01/25/88 13:18:52 PAGE 3

| 1 | 2 | 3 | 4 | *... .. 1 2 3 4 5 6 7 8 | | | | | | | | 9 |
|-------|------|--------|---|--|--|--|--|--|---|---|--|-------|
| SEQ | INST | OFFSET | GENERATED CODE | | | | | | | | | BREAK |
| 00001 | | | TITLE *IBM AS/400 PL/I | 5728PL1 R01M00 IRP LISTING FOR LP1413 | | | | | | ; | | |
| 00002 | | | ENTRY * (*7500000) EXT | | | | | | ; | | | |
| 00003 | | | DCL SPCPTR *76P0100 PARM | | | | | | ; | | | |
| 00004 | | | DCL SPCPTR *76P0200 PARM | | | | | | ; | | | |
| 00005 | | | DCL SPCPTR *76P0300 PARM | | | | | | ; | | | |
| 00006 | | | DCL SPCPTR *76P0400 PARM | | | | | | ; | | | |
| 00007 | | | DCL SPCPTR *76P0500 PARM | | | | | | ; | | | |
| 00008 | | | DCL SPCPTR *76P0600 PARM | | | | | | ; | | | |
| 00009 | | | DCL SPCPTR *76P0700 PARM | | | | | | ; | | | |
| 00010 | | | DCL SPCPTR *76P0800 PARM | | | | | | ; | | | |
| 00011 | | | DCL SPCPTR *76P0900 PARM | | | | | | ; | | | |
| 00012 | | | DCL SPCPTR *76P0A00 PARM | | | | | | ; | | | |
| 00013 | | | DCL SPCPTR *76P0B00 PARM | | | | | | ; | | | |
| 00014 | | | DCL SPCPTR *76P0C00 PARM | | | | | | ; | | | |
| 00015 | | | DCL SPCPTR *76P0D00 PARM | | | | | | ; | | | |
| 00016 | | | DCL SPCPTR *76P0E00 PARM | | | | | | ; | | | |
| 00017 | | | DCL SPCPTR *76P0F00 PARM | | | | | | ; | | | |
| 00018 | | | DCL SPCPTR *76P1000 PARM | | | | | | ; | | | |
| 00019 | | | DCL SPCPTR *76P1100 PARM | | | | | | ; | | | |
| 00020 | | | DCL SPCPTR *76P1200 PARM | | | | | | ; | | | |
| 00021 | | | DCL SPCPTR *76P1300 PARM | | | | | | ; | | | |
| 00022 | | | DCL SPCPTR *76P1400 PARM | | | | | | ; | | | |
| 00023 | | | DCL SPCPTR *76P1500 PARM | | | | | | ; | | | |
| 00024 | | | DCL SPCPTR *76P1600 PARM | | | | | | ; | | | |
| 00025 | | | DCL SPCPTR *76P1700 PARM | | | | | | ; | | | |
| 00026 | | | DCL SPCPTR *76P1800 PARM | | | | | | ; | | | |
| 00027 | | | DCL SPCPTR *76P1900 PARM | | | | | | ; | | | |
| 00028 | | | DCL SPCPTR *76P1A00 PARM | | | | | | ; | | | |
| 00029 | | | DCL SPCPTR *76P1B00 PARM | | | | | | ; | | | |
| 00030 | | | DCL SPCPTR *76P1C00 PARM | | | | | | ; | | | |
| 00031 | | | DCL SPCPTR *76P1D00 PARM | | | | | | ; | | | |
| 00032 | | | DCL SPCPTR *76P1E00 PARM | | | | | | ; | | | |
| 00033 | | | DCL SPCPTR *76P1F00 PARM | | | | | | ; | | | |
| 00034 | | | DCL SPCPTR *76P2000 PARM | | | | | | ; | | | |
| 00035 | | | DCL OL *7500000 (*76P0100,*76P0200,*76P0300,*76P0400,*76P0500,*76P0600,*76P0700,*76P0800,*76P0900,*76P0A00,*76P0B00,*76P0C00,*76P0D00,*76P0E00,*76P0F00,*76P1000,*76P1100,*76P1200,*76P1300,*76P1400,*76P1500,*76P1600,*76P1700,*76P1800,*76P1900,*76P1A00,*76P1B00,*76P1C00,*76P1D00,*76P1E00,*76P1F00,*76P2000) PARM EXT MIN(0) ; | | | | | | ; | | | |
| | | | /* DISPLAY ARRAY (FOR ACTIVE BLOCKS) */ | | | | | | ; | | | |
| 00036 | | | | | | | | | ; | | | |
| 00037 | | | DCL DD *DSPAREA CHAR(32) BDRY(16) AUTO | | | | | | ; | | | |
| 00038 | | | DCL SPCPTR *DISPLAY(2) POS(1) DEF(*DSPAREA) | | | | | | ; | | | |
| | | | /* QPGADE -- ARRAY DESCRIPTOR */ | | | | | | ; | | | |
| 00039 | | | | | | | | | ; | | | |
| 00040 | | | DCL MSPPTR *ADE@ | | | | | | ; | | | |
| 00041 | | | DCL DD *ADE BAS(*ADE@) CHAR(204) INT | | | | | | ; | | | |
| 00042 | | | DCL DD *ADEHDR DEF(*ADE) CHAR(14) | | | | | | ; | | | |
| 00043 | | | DCL DD *ADEDADE DEF(*ADE) POS(15) CHAR(190) | | | | | | ; | | | |
| 00044 | | | DCL DD *ADECHDR DEF(*ADEDADE) CHAR(10) | | | | | | ; | | | |
| 00045 | | | DCL DD *ADECD DEF(*ADECHDR) BIN(2) | | | | | | ; | | | |
| 00046 | | | DCL DD *ADETL DEF(*ADECD:BR) POS(3) BIN(4) | | | | | | ; | | | |
| 00047 | | | DCL DD *ADERVO DEF(*ADECHDR) POS(7) BIN(4) | | | | | | ; | | | |

Figure A-7. Examples of Compiler Debugging Information

The *LIST value for the GENOPT parameter causes printing of IRP and machine instructions when compilation ends. The headings in this IRP listing indicate the following information:

- 1** SEQ: A sequential numbering of the IRP statements. Error messages such as IRP syntax errors issued by the program resolution monitor use this number to refer to the IRP statements in error.

COMPILER DEBUGGING OPTIONS

- 2 INST: A sequential numbering of the machine instructions generated from the IRP statements. Not all IRP statements generate machine instructions. The instruction number can be used as a breakpoint for OS/400 debugging functions. See "Using Debug" on page 3-10, or the *Programming: Control Language Programmer's Guide* for information about breakpoints.
- 3 OFFSET: Displacement of the machine instruction into the instruction portion of the program template.
- 4 GENERATED CODE: Machine instructions that have been generated from IRP statements.
- 5 GENERATED OUTPUT: IRP statements.
- 6 BREAK: Breakpoints in the IRP that can be used for stopping points in OS/400 debugging functions. If the breakpoint is a number, it indicates the PL/I source statement that the IRP statement was generated from.

```

ODT ODT NAME          SEQ CROSS REFERENCE (* INDICATES WHERE DEFFINED)
7  8
00E3 .AU00001 248*
00E9 .AU00002 254*
00E7 .BA00001 252*
00ED .BA00002 258*
00E2 .BLK0001 247* 248 249 250 251 252 438 439 440 441 442 446 450 451 452 453 695 820 912
00E8 .BLK0002 253* 254 255 256 257 258 455 456 696 1125 1141
00CF .DBGPTR1 223*
00D0 .DBGPTR2 224*
00E5 .EX00001 250*
00EB .EX00002 256*
00E6 .PA00001 251*
00EC .PA00002 257*
0186 .STATIC0 424* 425 426 427 431 432 433 534 535 565 566
00E4 .ST00001 249*
00EA .ST00002 255*
019A .010000C 445*
019E .0100010 449*
0198 .020000A 443*
0199 .020000B 444*
019C .020000E 447*
019D .020000F 448*
0190 .0300012 434* 844 896 897 1137
0191 .0300013 435* 824 1136 1137
0192 .0300014 436* 825 843 844
0198 .090000D 446* 447 448 449 865 876 888
0197 .0900009 442* 443 444 445 875 880
018F .0900011 433* 434 435 436
018D .4700006 431* 826 827 832 849 864 866 867 887 889 890 899
018E .4700008 432* 828 829 858 879 881 882 904
0193 .7D00004 438* 837
01A3 .7D00015 455*
0187 .7E00000 425* 823
01F0 *ADDEXTC 601 624*
0026 *ADE 41* 42 43
0025 *ADE@ 40* 41
002E *ADEBDE 49* 50 51 52
002D *ADEBDS 48* 49
002A *ADECD 45*
0029 *ADECHDR 44* 45 46 47
0028 *ADEDADE 43* 44 48
0027 *ADEHDR 42*
002F *ADELB 50*
0031 *ADEMULT 52*
002C *ADERVO 47*
002B *ADETL 46*
0030 *ADEUB 51*
01DD *ALCSTAT 538 561*
01DE *ALSNOEX 568 570*
01DC *ALSNSI 560* 562 574
01F5 *AXCEND 649 653*
01F3 *AXCNPEC 646 650*
01EB *AXCNSI 619* 625 654
01F4 *AXCNTEC 644 653*
01EE *AXCPNAM 622* 650
01F2 *AXCPSET 627 636 638*
01F1 *AXCSAME 628 637*

```

The *XREF value for the GENOPT parameter causes printing of the cross-reference listing. The headings in this listing include:

- 7** ODT: The variable number assigned by the PRM to each variable of the IRP.
- 8** ODT NAME: The name of the variable in IRP.

COMPILER DEBUGGING OPTIONS

9 SEQ CROSS REFERENCE (* INDICATES WHERE DEFINED): The lines in the IRP listing where the variable is referenced. The * indicates where the declaration for the variable is found.



| 5728SS1 R01 M00 880715 | GENERATED OUTPUT | | | | | | | | 01/25/88 13:18:52 | PAGE | 3 |
|------------------------|------------------|----------|----------|----------|----------|----------|----------|----------|-------------------|----------|---|
| OFFSET | MI | TEMPLATE | DISPLAY | | | | | | | | |
| 00000000 | 000035DC | 00000000 | 0201D3D7 | F1F4F1F3 | 40404040 | 40404040 | 40404040 | 40404040 | 40404040 | 40404040 | |
| 00000020 | 40404040 | 40404040 | 80000000 | 00000000 | 0000015F | 00010000 | 00000000 | 00000000 | 00000000 | 00000000 | |
| 00000040 | 00000000 | 00000000 | 00542A2F | 5E000400 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | |
| 00000060 | 668040FC | 00000000 | 00000000 | 02120270 | 00000100 | 0000107C | 00001A40 | 00000009 | 00000000 | 00000000 | |
| 00000080 | 000001B8 | 00003424 | 00000000 | 00000073 | 000026B0 | 00000000 | 00000212 | 00000000 | 00000000 | 00000000 | |
| 000000A0 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | |
| 000000C0 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | |
| 000000E0 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | 00000000 | |
| 00000100 | 00000F7C | 229301CD | 000000CC | 3C46C000 | 00812000 | 01BB1042 | 00812001 | 302200CD | 00000000 | 00000000 | |
| 00000120 | 01BC1011 | 40E00081 | 22930202 | 000000CC | 22A10000 | 213201CC | 00CC10BE | 00232000 | 00000000 | 00000000 | |
| 00000140 | 054701C5 | 00002001 | 013201CB | 01C60083 | 01BF01BE | 20001CC2 | C00001C1 | 01CE01CF | 00000000 | 00000000 | |
| 00000160 | 00830068 | 01C22010 | 1CE24000 | 007F0000 | 01CF1CD2 | C000007F | 01CB01CF | 0132007F | 00000000 | 00000000 | |
| 00000180 | 00001011 | 01D22283 | 01C30000 | 00000083 | 006801C2 | 20100132 | 008E01CB | 054701C5 | 00000000 | 00000000 | |
| 000001A0 | 00002000 | 1042008A | 01C91042 | 008B01C8 | 025201C4 | 0000107B | 01CA01D0 | 01D120B3 | 00000000 | 00000000 | |
| 000001C0 | 00660074 | 20001197 | 00952040 | 002201A5 | 01CB0083 | 01A901A5 | 01A81CE2 | 40000080 | 00000000 | 00000000 | |
| 000001E0 | 000001D3 | 00830186 | 00802000 | 00830098 | 01862000 | 10B2009B | 20801011 | 01CC2293 | 00000000 | 00000000 | |
| 00000200 | 01DD0000 | 00CC3011 | 01CC2132 | 01D400CC | 114300A6 | 200F1193 | 00A701D9 | 1C469000 | 00000000 | 00000000 | |
| 00000220 | 00A600A8 | 01DB1043 | 01D500A6 | 3FFF1193 | 01D601DA | 114701D5 | 20A00032 | 01D700A4 | 00000000 | 00000000 | |
| 00000240 | 006201D7 | 01D51042 | 00A801D5 | 301101D4 | 213201DC | 00CC0082 | 005D01BA | 008300A4 | 00000000 | 00000000 | |
| 00000260 | 006D2000 | 00830186 | 006C00A6 | 00830098 | 01862000 | 114300A6 | 005F1C46 | 900000A6 | 00000000 | 00000000 | |
| 00000280 | 00A801DE | 029301D8 | 000000CC | 30BE0099 | 200010B2 | 009A0061 | 10B2009B | 00620293 | 00000000 | 00000000 | |
| 000002A0 | 01E20000 | 00CC3011 | 01DC2132 | 01DF00CC | 0082009F | 01880082 | 00A20189 | 008300AA | 00000000 | 00000000 | |
| 000002C0 | 005D0065 | 104201E0 | 20013C46 | 100001E0 | 006001EA | 008300C5 | 00772000 | 008300BB | 00000000 | 00000000 | |
| 000002E0 | 00C500CB | 104201E1 | 20013CC2 | 400000BF | 00AC01E7 | 1C464000 | 01E100CA | 01E51C46 | 00000000 | 00000000 | |
| 00000300 | A00001E1 | 00C801E6 | 008300BB | 00BB00C4 | 114301E1 | 20011011 | 01E43C46 | 400000C7 | 00000000 | 00000000 | |
| 00000320 | BFFF01E6 | 008300C5 | 007000C7 | 008300BB | 00C500CB | 104201E1 | 20011011 | 01E42293 | 00000000 | 00000000 | |
| 00000340 | 01F00000 | 00CC20B3 | 00A30070 | 00BD10B2 | 00A02080 | 101101E9 | 208300A3 | 007000BD | 00000000 | 00000000 | |
| 00000360 | 10B200A0 | 20001CC2 | C00000C2 | 00B801E8 | 1CC2C000 | 00BF0061 | 01E80083 | 00500070 | 00000000 | 00000000 | |
| 00000380 | 00BD0083 | 00530098 | 200020B3 | 00AA00AA | 00BA0083 | 009F009F | 200100B3 | 00A200A2 | 00000000 | 00000000 | |
| 000003A0 | 20101143 | 01E02001 | 101101E3 | 301101DF | 213201EB | 00CC0083 | 00A40071 | 20001C46 | 00000000 | 00000000 | |
| 000003C0 | 400000C8 | 200001F2 | 1C462000 | 00C800CA | 01F11042 | 00C700A6 | 009200C5 | 00A61143 | 00000000 | 00000000 | |
| 000003E0 | 00A62200 | 029301D8 | 000000CC | 304200C7 | BFFF0083 | 00BB00C5 | 00CB1042 | 00C82000 | 00000000 | 00000000 | |
| 00000400 | 101101F2 | 208300BB | 00BB00C4 | 30B200BE | 00AB1042 | 00BD00A6 | 114300BB | 00C01143 | 00000000 | 00000000 | |
| 00000420 | 00A600C1 | 114300A6 | 00C00293 | 01D80000 | 00CC3CC2 | C00000C2 | 00B801F4 | 00830050 | 00000000 | 00000000 | |
| 00000440 | 007000BD | 1CC2C000 | 00BF0061 | 01F30132 | 005201CB | 00830053 | 00982000 | 101101F5 | 00000000 | 00000000 | |
| 00000460 | 30B301E6 | 00BF2040 | 01640052 | 01EC0000 | 00000132 | 00530000 | 314300C8 | 20011011 | 00000000 | 00000000 | |
| 00000480 | 01EB2132 | 01F600CC | 008300A4 | 006F2000 | 114300A6 | 200F1193 | 00A701F9 | 104201F7 | 00000000 | 00000000 | |
| 000004A0 | 00A61143 | 00A60034 | 1C469000 | 00A600A8 | 01FA0293 | 01D80000 | 00CC20B3 | 0038006E | 00000000 | 00000000 | |
| 000004C0 | 01F700A2 | 003A0076 | 10B2003C | 00350132 | 004100CD | 01320044 | 01CB0022 | 00450098 | 00000000 | 00000000 | |
| 000004E0 | 01320046 | 00000132 | 00474024 | 00900132 | 00480000 | 10B20049 | 00361042 | 004A0090 | 00000000 | 00000000 | |
| 00000500 | 10B2004B | 003710B2 | 004C00A6 | 10BE004D | 20000022 | 00760038 | 01324024 | 00900076 | 00000000 | 00000000 | |
| 00000520 | 11930095 | 20BF1011 | 01F62132 | 01FB00CC | 1042003B | 008320B3 | 0038006E | 00380083 | 00000000 | 00000000 | |
| 00000540 | 4024004A | 00382000 | 1C461000 | 003B2000 | 01FD0132 | 00E24024 | 20010132 | 00E84024 | 00000000 | 00000000 | |
| 00000560 | 20020083 | 00380076 | 20001011 | 01FB2132 | 01FE00CC | 00830038 | 00762000 | 00920076 | 00000000 | 00000000 | |
| 00000580 | 003A0132 | 4024004A | 00470132 | 00CD0041 | 008300A4 | 006F2000 | 00A200A6 | 00381011 | 00000000 | 00000000 | |
| 000005A0 | 01FE2132 | 020000CC | 1CE6C000 | 00760075 | 02041C46 | 40000097 | 20000203 | 10420086 | 00000000 | 00000000 | |
| 000005C0 | 02010283 | 40670201 | 000000DA | 213201C2 | 00880132 | 01C10089 | 1197008D | 20803011 | 00000000 | 00000000 | |
| 000005E0 | 02002083 | 00380076 | 20000083 | 0205006E | 003A1CE6 | C0000205 | 00750208 | 00824085 | 00000000 | 00000000 | |
| 00000600 | 20010206 | 10420086 | 00D60283 | 406700D6 | 000000DA | 30220041 | 02091011 | 40E1004A | 00000000 | 00000000 | |
| 00000620 | 3C2A4000 | 003F2080 | 02070293 | 02020000 | 00CC22A1 | 20012083 | 00380076 | 20001C2A | 00000000 | 00000000 | |
| 00000640 | 2000003E | 2040020B | 10220041 | 020A3011 | 40E1004A | 30B2020C | 00841042 | 0086020D | 00000000 | 00000000 | |
| 00000660 | 02834067 | 020D0000 | 00DA10B2 | 0084020C | 054701C5 | 00002003 | 01320210 | 01C703E1 | 00000000 | 00000000 | |
| 00000680 | 020E3042 | 0086020D | 02834067 | 020D0000 | 00DA30B2 | 010F00F6 | 1042010E | 204B0293 | 00000000 | 00000000 | |
| 000006A0 | 021E0000 | 01053011 | 00CC30B2 | 010F00FF | 1042010E | 204A0293 | 021E0000 | 01053011 | 00000000 | 00000000 | |
| 000006C0 | 00CC30B2 | 010F00F7 | 1042010E | 204B0293 | 021E0000 | 01053011 | 00CC30B2 | 010F00FA | 00000000 | 00000000 | |
| 000006E0 | 1042010E | 204B0293 | 021E0000 | 01053011 | 00CC3042 | 010E2048 | 0293021E | 00000105 | 00000000 | 00000000 | |

The *DUMP value for the GENOPT parameter causes printing of the program template. This template shows:

- 10** OFFSET: The displacement from the beginning of the program template of this line of data. The offset is displayed in hexadecimal notation.
- 11** MI TEMPLATE DISPLAY: The data contained in the program template at the offset specified. The data is displayed in hexadecimal notation.

Using the SERVICE Parameter

You can specify SERVICE(*YES) on the CRTPLIPGM command to access facilities to debug the PL/I compiler in batch or interactive mode.

You can debug in batch mode by creating a source file member to contain the debugging commands you want to use. You must name this source file member QPLIDBGINP, and place it in the same library as the program source file.

You can debug interactively in any of the following ways:

- Debug in batch mode by entering debugging commands in source file member QPLIDBGINP
- Debug interactively by entering debugging commands from your work station
- Use a combination of the above two methods

SERVICE Debugging Commands

The commands that you can use to debug the compiler are shown in the following table, and described in the text below.

You can specify these commands either from your terminal, from source file member QPLIDBGINP, or from either place, as shown below:

| Command | Use from QPLIDBGINP | Use from Work Station |
|------------------|---------------------|-----------------------|
| * | Yes | No |
| INSERT or INS | Yes | Yes |
| TERMINATE or TER | Yes | Yes |
| DEBUG or DEB | Yes | Yes |
| NODEBUG or NOD | Yes | Yes |
| EXECUTE or EXE | Yes | Yes |
| NOEXECUTE or NOE | Yes | Yes |

Figure A-8 (Part 1 of 2). Commands to use with the SERVICE parameter

COMPILER DEBUGGING OPTIONS

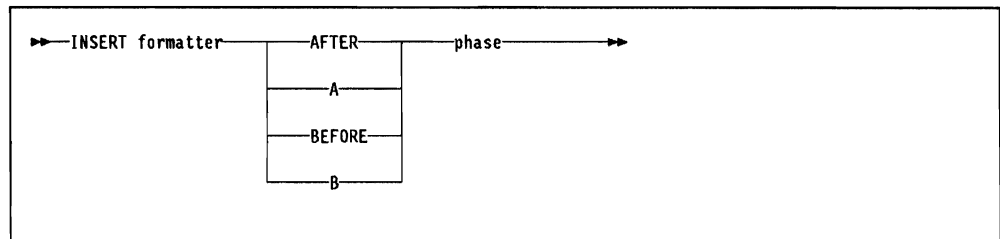
| Command | Use from QPLIDBGINP | Use from Work Station |
|----------------|---------------------|-----------------------|
| CONSOLE or CON | Yes | No |
| SEGMENT or SEG | No | Yes |
| ? | No | Yes |

Figure A-8 (Part 2 of 2). Commands to use with the SERVICE parameter

The command syntax and descriptions follow:

*

Use in file QPLIDBGINP to indicate that text following the asterisk is treated as comments.



The formatter will follow or precede the phase.

Formatters and phases must be abbreviated to three characters as follows:

1. Drop the first three characters of the phase name (QPC).
2. Retain the fourth and fifth characters of the phase name.
3. Drop the first two digits (00) of the three digit number.
4. Retain the last digit.

For example, the formatter QPCPF008 must be abbreviated as PF8, and the phase QPCLP001 must be abbreviated as LP1.

The following table lists the places where you can use a formatter, and the type of intermediate text that is formatted and listed.

Note: ZZ1 is a phase, but it also acts as a formatter.

COMPILER DEBUGGING OPTIONS

| Formatter | BEFORE or AFTER | Phase | Type of Text |
|-----------|---|---|---|
| ZZ1 | BEFORE AFTER | any | |
| PF8 | AFTER AFTER AFTER AFTER AFTER AFTER AFTER AFTER AFTER AFTER AFTER AFTER AFTER AFTER AFTER AFTER AFTER | HP1 DB1 DX1 DY1 EA1 GC1 AG1 AD1 DI1 ST1 RC1 FL1 TR1 SK1 SA1 | Polish-1 Polish-1 Polish-1 Polish-1 N-Address-1 N-Address-1 N-Address-1 N-Address-1 N-Address-1 N-Address-1 N-Address-1 N-Address-1 FIT EFIT EFIT |
| PF9 | AFTER AFTER AFTER AFTER AFTER AFTER AFTER | HP1 DB1 DX1 GC1 AG1 AD1 RC1 | Polish-2 Polish-2 Polish-2 N-Address-2 N-Address-2 N-Address-2 N-Address-2 |

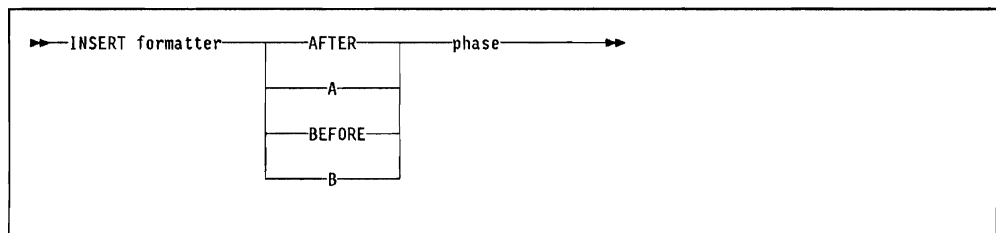
Figure A-9 (Part 1 of 2). Using the INSERT Command

COMPILER DEBUGGING OPTIONS

| Formatter | BEFORE or AFTER | Phase | Type of Text |
|-----------|-----------------------|------------|--------------------|
| DF3 | AFTER | DB1 | Dictionary |
| | AFTER | DX1 | Dictionary |
| | AFTER | DY1 | Dictionary |
| | AFTER | EA1 | Dictionary |
| | AFTER | GC1 | Dictionary |
| | AFTER | AG1 | Dictionary |
| | AFTER | AD1 | Dictionary |
| | AFTER | DI1 | Dictionary |
| | AFTER | ST1 | Dictionary |
| | AFTER | RC1 | Dictionary |
| | AFTER | FL1 | Dictionary |
| | AFTER | TR1 | Dictionary |
| | AFTER | SK1 | Dictionary |
| | AFTER | SA1 | Dictionary |
| | AFTER | DC1 | Dictionary |
| AFTER | CG1 | Dictionary | |
| AFTER | FA1 | Dictionary | |
| AFTER | TM1 | Dictionary | |
| TF1 | AFTER | LP1 | Tokenized Text |
| F1 | AFTER | LP1 | Constant string |
| | AFTER | DX1 | Constant string |
| | AFTER | AG1 | Constant string |
| | AFTER | AD1 | Constant string |
| | AFTER | DI1 | Constant string |
| | AFTER | FL1 | Constant string |
| | AFTER | TR1 | Constant string |
| | AFTER | SK1 | Constant string |
| | AFTER | SA1 | Constant string |
| | AFTER | DC1 | Constant string |

Figure A-9 (Part 2 of 2). Using the INSERT Command

The first time you call ZZ1, (by specifying SERVICE (*YES) on the CRTPLIPGM command), source file member QPLIDBGINP is processed, if it exists. If you call ZZ1 again, (either from QPLIDBGINP or interactively from your work station), you may only process commands from your work station. Therefore, if you call ZZ1 from QPLIDBGINP, processing is temporarily halted to enable you to enter any debugging commands from your work station.



processes the normal ending of compilation following or preceding the specified phase. Phase TM1 is called.

DEBUG phase

This puts you in debug mode (the equivalent of issuing the STRDBG CL command), with a breakpoint at the beginning of the program. For information about using debug mode, refer to *Programming: Control Language Programmer's Guide*.

NODEBUG phase

If you have previously specified DEBUG for a phase, you can now specify that you do not want to enter debug mode for that phase.

NOEXECUTE phase

The phase specified will not be processed.

EXECUTE phase

If you have previously specified NOEXECUTE for a phase, you can now specify that the phase be processed.

CONSOLE

Specifies, from source file member QPLIDBGINP, that commands may be entered from the work station.

SEGMENT segment-identifier

produces a dump of the named segment-identifier. Segment-identifier is a number from 1 through 19.

?

In interactive mode, a help text screen shows the syntax and definitions of the above commands.

Quantitative Limits of Compiler

The following table gives information about the maxima, minima, and defaults of the AS/400 PL/I compiler.

General

| | |
|---|--------|
| Collating sequence | EBCDIC |
| Digits in exponent of floating-point variable | 3 |
| Length of string returned by TIME | 9 |

Maxima

| | |
|--|--------|
| %INCLUDE – number of members | 20 |
| %SKIP – number of lines | 99 |
| A,B,B1,B4,X,COL | 32 767 |
| Arguments for MAX/MIN built-in functions | 2 |
| Arguments in a CALL statement | 32 |
| Arguments in a function reference | 31 |
| BINARY FIXED precision | 31 |

QUANTITATIVE LIMITS OF COMPILER

Maxima

| | |
|---|--|
| Blocks in an external procedure | 255 |
| DECIMAL FIXED precision | 15 |
| Depths of nesting | |
| %INCLUDE source file | 64 |
| Attribute factorization | 1 |
| Comments | 1 |
| Combined nesting depth (see glossary) | 200 |
| DO in GET and PUT statements | 49 |
| Do-groups | 49 |
| Functions | 50 |
| IF...THEN...ELSE statements | 49 |
| INITIAL attribute iterations | 1 |
| Procedures and begin-blocks | 50 |
| SELECT statements | 49 |
| Structures | 15 |
| Dimensions of an array | 15 |
| Exponent in BINARY FLOAT variables | |
| Short form | 127 |
| Long form | 1023 |
| Exponent in DECIMAL FLOAT variables | |
| Short form | 38 approximately |
| Long form | 308 approximately |
| Extent of an array | 32 767 |
| INITIAL attribute iteration factor | 32 767 |
| Keylength (composite key) | 120 |
| Labels per single statement | 1 |
| Length of array | 4194303 bytes |
| Length of BIT string | 32 767 bits |
| Length of CHARACTER VARYING data item | 32 765 characters |
| Length of fully qualified name (excluding periods) | 15 structures (levels) * 31 characters |
| Length of title | 33 characters |
| Length of programmer-defined name | 31 characters |
| Length of programmer-defined AS/400 name | 10 characters |
| Length of statement | 32 767 characters |
| Level number in structure | 255 |
| LINE expression | 32 767 |
| Linesize — record format F | 32 765 |
| Names defined by programmer | |
| if all names are 31 characters long | 1760 |
| if all names are 10 characters long | 4080 |
| On-units in a block | 50 |
| Pagesize | 32 767 |
| Parameters in a procedure | 32 |
| PICTURE format specification — length | 255 characters |
| PICTURE — number of digits in specification | 15 |
| Record size — record format F | 32 767 characters |

QUANTITATIVE LIMITS OF COMPILER

Maxima

| | |
|---------------------------------------|-------------------|
| Significand of BINARY FLOAT variable | |
| Short form | 24 binary digits |
| Long form | 53 binary digits |
| Significand of DECIMAL FLOAT variable | |
| Short form | 7 decimal digits |
| Long form | 16 decimal digits |
| SKIP expression | 32 767 |
| Source records | 99999 |
| Statements in a program | 9999 |
| Substatements in a statement | 9999 |

Minima

| | |
|-----------------------------------|---|
| Array extent | 1 |
| KEYDISP integer constant | 0 |
| Length of bit or character string | 0 |
| Linesize – record format F | 1 |
| PAGESIZE expression | 1 |
| Relative record number | 1 |
| Scale factor for FIXED variables | 0 |
| SKIP expression | 1 |
| %SKIP – number of lines | 1 |

Defaults

| | |
|-----------------------------------|------|
| %SKIP – number of lines | 1 |
| BIT string length | 1 |
| CHARACTER string length | 1 |
| FIXED BINARY precision and scale | 31,0 |
| FIXED DECIMAL precision and scale | 5,0 |
| FLOAT DECIMAL precision | 7 |
| FLOAT BINARY precision | 24 |
| LINE | 1 |
| PAGESIZE | 60 |
| PRINT file format | F |
| SKIP expression | 1 |

QUANTITATIVE LIMITS OF COMPILER



Appendix B. The AS/400 PL/I Language Summary and Character Set

The AS/400 PL/I compiler was designed to conform to the General-Purpose Subset (Subset G) of PL/I as defined by the American National Standards Institute (ANSI).

The following table compares the language features of AS/400 PL/I to the ANSI Subset G standard. The table shows only how AS/400 PL/I deviates from Subset G.

Following the table is a listing of the PL/I character set.

For details about specific language features, refer to the appropriate section in this manual.

| Key: | |
|-------|--|
| = | The AS/400 implementation is the same as defined by ANSI Subset G. |
| A | The AS/400 implementation offers additional function above ANSI Subset G. |
| R | The AS/400 implementation has restrictions below what is defined by ANSI Subset G. |
| N | The AS/400 implementation does not support this ANSI Subset G language feature. |
| I | This language feature is implementation defined (in conformance with ANSI Subset G). |
| blank | This language feature is not in Subset G . |
| S | For features where A or blank appear under the Subset G column, indicates the source that the AS/400 PL/I feature was modelled after: <ul style="list-style-type: none">• the feature is part of the full ANSI PL/I language, but is not in Subset G.• the feature is a language extension that also exists in other IBM PL/I compilers.• the feature is a language extension that exists only in the AS/400 PL/I compiler. |

| Statements | | | | |
|--|---------------------------------|-----------|------------------|------------------|
| LANGUAGE FEATURE | Subset G | Full ANSI | IBM Extension | AS/400 Extension |
| ALLOCATE | = | | | |
| Assignment BY NAME scalar to connected structure | A | | S S | |
| BEGIN | = | | | |
| CALL shortened argument list | A | | S | |
| CLOSE | = | | | |
| DECLARE (see Attributes) factored attributes nested factoring declaration of SYSIN and SYSPRINT assumed | R R N | | S | |
| DELETE KEY OPTIONS | A A | | | S S |
| DO | = | | | |
| DO iterate control variable BY TO UNTIL | A A A A | | S S S S | |
| END | = | | | |
| FORMAT | N | | | |
| FREE | = | | | |
| GET FILE LIST STRING format items B2/B3 P 'picture' | R A N N R N N | | S | |
| GOTO or GO TO | = | | | |
| IF THEN/ELSE unit labeled | A | | S | |

| LANGUAGE FEATURE | Subset G | Full ANSI | IBM Extension | AS/400 Extension |
|--|---|-----------|---------------|------------------|
| ITERATE | | | S | |
| LEAVE | | | S | |
| null | = | | | |
| ON SNAP SYSTEM | A | | S S | |
| OPEN DIRECT ENVIRONMENT KEYED PRINT RECORD SEQUENTIAL STREAM TAB TITLE other than character string | R N N N N N N N N R N | | | |
| PROCEDURE | = | | | |
| PUT FILE LIST STRING format items B2/B3 P 'picture' TAB(n) | R A N N R N N N N | | S | |
| READ KEY KEYTO OPTIONS | A A A | | | S S S |
| RETURN descriptor FILE | R R N | | | |
| REVERT | N | | | |
| REWRITE KEY OPTIONS | A A | | | S S |
| SELECT | | | S | |

| LANGUAGE FEATURE | Subset G | Full ANSI | IBM Extension | AS/400 Extension |
|-----------------------------|----------|-----------|---------------|------------------|
| SIGNAL | = | | | |
| STOP | = | | | |
| WRITE KEYFROM OPTIONS | A A | | | S S |

| Directives | | | | |
|------------------|----------|-----------|---------------|------------------|
| LANGUAGE FEATURE | Subset G | Full ANSI | IBM Extension | AS/400 Extension |
| %INCLUDE | A | | | s |
| %PAGE | | | S | |
| %PROCESS | | | S | |
| %SKIP | | | S | |

| Label Prefixes | | | | |
|----------------------------|----------|-----------|---------------|------------------|
| LANGUAGE FEATURE | Subset G | Full ANSI | IBM Extension | AS/400 Extension |
| FORMAT | N | | | |
| PROCEDURE | = | | | |
| THEN/ELSE unit | | | S | |
| WHEN/OTHERWISE unit | | | | S |
| other statements | = | | | |
| single label only | = | | | |
| subscripted label prefixes | N | | | |

| Conditions | | | | |
|--|-----------------------|----------------------|--------------------------|-----------------------------|
| LANGUAGE FEATURE | Subset G | Full ANSI | IBM Extension | AS/400 Extension |
| on units FIXEDOVERFLOW OVERFLOW TRANSMIT UNDERFLOW ZERODIVIDE | R N N N N | S | | |
| default enablement RECORD STORAGE | A | S S | | |

| Built-in Functions | | | | |
|---------------------------|-----------------|------------------|----------------------|-------------------------|
| LANGUAGE FEATURE | Subset G | Full ANSI | IBM Extension | AS/400 Extension |
| ABS | = | | | |
| ACOS | = | | | |
| ADDR | = | | | |
| ASIN | = | | | |
| ATAN (x[,y]) | = | | | |
| ATAND (x[,y]) | = | | | |
| ATANH | = | | | |
| BINARY | = | | | |
| BIT (v[,1]) | = | | | |
| BOOL | N | | | |
| CEIL | N | | | |
| CHARACTER (v[,1]) | = | | | |
| COLLATE | N | | | |
| COPY | = | | | |
| COS | = | | | |
| COSD | = | | | |
| COSH | = | | | |
| DATE | = | | | |
| DECIMAL | = | | | |
| DIMENSION | = | | | |
| DIVIDE | = | | | |
| EXP | = | | | |
| FIXED | = | | | |
| FLOAT | = | | | |
| FLOOR | N | | | |
| HBOUND | = | | | |
| INDEX | = | | | |
| LBOUND | = | | | |
| LENGTH | = | | | |

| LANGUAGE FEATURE | Subset G | Full ANSI | IBM Extension | AS/400 Extension |
|---------------------|----------|-----------|---------------|------------------|
| LINENO | = | | | |
| LOG | = | | | |
| LOG2 | = | | | |
| LOG10 | = | | | |
| MAX (x,y) | = | | | |
| MIN (x,y) | = | | | |
| MOD | = | | | |
| NULL | = | | | |
| ONCODE | = | | | |
| ONFILE | = | | | |
| ONKEY | = | | | |
| PAGENO | N | | | |
| PLIRETV | | | | S |
| PLISHUTDN | | | | S |
| ROUND | = | | | |
| SAMEKEY | | | S | |
| SIGN | = | | | |
| SIN | = | | | |
| SIND | = | | | |
| SINH | = | | | |
| SQRT | = | | | |
| STORAGE | | | S | |
| STRING | N | | | |
| SUBSTR (s,i[,j]) | = | | | |
| TAN | = | | | |
| TAND | = | | | |
| TANH | = | | | |
| TIME | = | | | |
| TRANSLATE (s,r[,t]) | = | | | |

| LANGUAGE FEATURE | Subset G | Full ANSI | IBM Extension | AS/400 Extension |
|------------------|----------|-----------|---------------|------------------|
| TRUNC | = | | | |
| UNSPEC | = | | | |
| VALID | N | | | |
| VERIFY | = | | | |

| Built-in Subroutines | | | | |
|----------------------|----------|-----------|---------------|------------------|
| LANGUAGE FEATURE | Subset G | Full ANSI | IBM Extension | AS/400 Extension |
| PLICOMMIT | | | | S |
| PLIDUMP | | | S | |
| PLIIOFDB | | | | S |
| PLIOPNFDB | | | | S |
| PLIRCVMSG | | | | S |
| PLIRETC | | | S | |
| PLIROLLBACK | | | | S |

| Pseudovariabes | | | | |
|------------------|----------|-----------|---------------|------------------|
| LANGUAGE FEATURE | Subset G | Full ANSI | IBM Extension | AS/400 Extension |
| PAGENO | N | | | |
| STRING | N | | | |
| SUBSTR (s,i[,j]) | = | | | |
| UNSPEC | = | | | |

| Attributes | | | | |
|---|---------------------------------|-----------|---------------|------------------|
| LANGUAGE FEATURE | Subset G | Full ANSI | IBM Extension | AS/400 Extension |
| ALIGNED BIT ALIGNED attribute optional | R R N | | | |
| AUTOMATIC length/dimension reference subscripted pointer qualified expression | R R R N N N N | | | |
| BASED length/dimension reference expression | R R N N | | | |
| BINARY | = | | | |
| BIT | = | | | |
| BUILTIN | = | | | |
| CHARACTER | = | | | |
| constant bit B2/B3 FORMAT LABEL constant dimension array | R R N N R N | | | |
| DECIMAL | = | | | |
| DEFINED | N | | | |
| dimension lower/upper bound: reference subscripted pointer qualified expression lower bound not = 1 | R R R N N N N | | | |
| DIRECT | = | | | |

| LANGUAGE FEATURE | Subset G | Full ANSI | IBM Extension | AS/400 Extension |
|---|-----------------------|-----------|---------------|------------------|
| ENTRY descriptor asterisk shortened argument list | A A | | S S | |
| ENVIRONMENT | I | | | |
| EXTERNAL identifier declared more than once in external procedure | R N | | | |
| FILE variable/parameter | R N | | | |
| FIXED | = | | | |
| FLOAT | = | | | |
| INITIAL reference (only NULL built-in) iteration factor with inherited dimensions | R R R N | | | |
| INPUT | = | | | |
| INTERNAL | = | | | |
| KEYED | = | | | |
| LABEL | = | | | |
| length reference subscripted pointer qualified expression | R R N N N | | | |
| member length/dimension reference expression | R R N N | | | |
| non-varying | = | | | |
| OPTIONS | I | | | |
| OUTPUT | = | | | |
| parameter | = | | | |
| PICTURE R character | A | S | | |

| LANGUAGE FEATURE | Subset G | Full ANSI | IBM Extension | AS/400 Extension |
|---|---------------------|----------------------|--------------------------|-----------------------------|
| POINTER | = | | | |
| precision | = | | | |
| PRINT | = | | | |
| real | = | | | |
| RECORD | = | | | |
| RETURNS descriptor FILE | R R N | | | |
| SEQUENTIAL | = | | | |
| STATIC | = | | | |
| STREAM | = | | | |
| structure | = | | | |
| UNALIGNED allowed for fixed binary and binary/decimal float | A | | S | |
| UPDATE | = | | | |
| VARIABLE | = | | | |
| VARYING | = | | | |

| References | | | | |
|---------------------------------------|----------|-----------|---------------|------------------|
| LANGUAGE FEATURE | Subset G | Full ANSI | IBM Extension | AS/400 Extension |
| unsubscripted | = | | | |
| subscripted interleaved subscripts | R N | | | |
| pointer qualified | = | | | |

| Character Set | | | | |
|-------------------------|----------|-----------|---------------|------------------|
| LANGUAGE FEATURE | Subset G | Full ANSI | IBM Extension | AS/400 Extension |
| letters A - Z | = | | | |
| letters a - z | | | S | |
| letters \$, #, @ | | | S | |
| digits 0 - 9 | = | | | |
| + - * / | = | | | |
| ., '_ blank | = | | | |
| ; : > < = & ~ () % | = | | | |
| extralingual characters | I | | | |

| Abbreviations | | | | |
|---------------------------|-----------------|------------------|----------------------|-------------------------|
| LANGUAGE FEATURE | Subset G | Full ANSI | IBM Extension | AS/400 Extension |
| AFT < AFTER > | | | | S |
| ALLOC < ALLOCATE > | = | | | |
| AUTO < AUTOMATIC > | = | | | |
| BFR < BEFORE > | | | | S |
| BIN < BINARY > | = | | | |
| CHAR < CHARACTER > | = | | | |
| COL < COLUMN > | = | | | |
| DCL < DECLARE > | = | | | |
| DEC < DECIMAL > | = | | | |
| DIM < only form allowed > | = | | | |
| ENV < ENVIRONMENT > | = | | | |
| EQL < EQUAL > | | | | S |
| EXT < EXTERNAL > | = | | | |
| FOFL < FIXEDOVERFLOW > | N | | | |
| INIT < INITIAL > | = | | | |
| INT < INTERNAL > | = | | | |
| NXT < NEXT > | | | | S |
| OFL < OVERFLOW > | N | | | |
| OTHER < OTHERWISE > | | | S | |
| PIC < PICTURE > | = | | | |
| PROC < PROCEDURE > | = | | | |
| PRV < PREVIOUS > | | | | S |
| PTR < POINTER > | = | | | |
| SEQL < SEQUENTIAL > | = | | | |
| STG < STORAGE > | | | S | |
| UFL < UNDERFLOW > | N | | | |
| UNAL < UNALIGNED > | = | | | |
| UNDF < UNDEFINEDFILE > | = | | | |
| VAR < VARIABLE > | = | | | |
| ZDIV < ZERODIVIDE > | N | | | |

The PL/I Character Set

The **language characters** are alphabetic characters, digits, and special characters.

There are 29 **alphabetic characters**:

| Character | Meaning |
|-----------|------------------|
| A - Z | English alphabet |

| IBM Extension | |
|---------------|----------------------|
| \$ | currency symbol |
| # | number sign |
| @ | commercial "at" sign |

End of IBM Extension

There are ten **digits**: 0 through 9.

There are 20 **special characters**:

| Character | Name |
|-----------|---------------------------------|
| | blank |
| = | equal sign or assignment symbol |
| + | plus |
| - | minus |
| * | asterisk or multiplication |
| / | slash or division |
| (| left parenthesis |
|) | right parenthesis |
| , | comma |
| . | point or period |
| ' | apostrophe |
| % | percent |
| ; | semicolon |
| : | colon |
| ¬ | not |
| & | and |
| | or |
| > | greater than |
| < | less than |
| - | break character |

You can combine certain special characters to create the following ten **composite symbols**:

| Symbol | Name |
|--------|--------------------------|
| < = | less than or equal to |
| | concatenation |
| ** | exponentiation |
| ¬ < | not less than |
| ¬ > | not greater than |
| ¬ = | not equal to |
| > = | greater than or equal to |
| /* | begin comment |
| */ | end comment |
| -> | points to |

EBCDIC Codes

The language characters, with their EBCDIC codes represented in hexadecimal notation can be found in Appendix E, "EBCDIC CODES."

The EBCDIC codes may be represented by different characters on different terminals or printers.

Lowercase Characters

You can use lowercase characters when writing a source program. In a comment or a character literal, the lowercase character maintains its identity as lowercase. In other uses (such as keywords or names), the lowercase character is equivalent to its corresponding uppercase character.

Extralingual Characters

Character constants and comments can contain any of the 256 EBCDIC codes. Any character that is not a language character is an **extralingual character**.

PL/I CHARACTER SET



Appendix C. Valid Combinations of Options for Input/Output Statements

The tables in this appendix show the combinations of options you can use in input and output statements. The following abbreviations are used in the table headings:

| | |
|---|--------|
| I | Input |
| O | Output |
| U | Update |

In the body of the tables, the following set of symbols is used:

| | |
|---|-----------|
| R | Required |
| O | Optional |
| – | Not valid |

Footnotes for all the tables are at the end of the appendix.

The following should be noted:

- Options that are not valid either produce an error or are ignored.
- You cannot specify the `OPTIONS` option in your input/output statements if you specify the `ENVIRONMENT` option `BLOCK` in your file declaration.
- If a statement is not listed for a particular file type, it cannot be used for that file type.
- Some input/output options that are listed as being invalid for certain file types will not produce error messages at compile time, but when running is attempted the statement will fail. The reason for this is that at compile time it is not known what AS/400 file type the file declared will be attached to.

Data Base Files with CONSECUTIVE organization

| | FILE ACCESS | | | | | | | | |
|----------------------|-----------------|---|-----------------|-------------------|----------------|-------------------|--------|----------------|----------------|
| | SEQUENTIAL | | | SEQUENTIAL KEYED | | | DIRECT | | |
| | I | O | U | I | O | U | I | O | U |
| READ | | | | | | | | | |
| FILE | R | - | R | R | - | R | R | - | R |
| INTO SET | R | - | R | R | - | R | R | - | R |
| KEY | - | - | - | O ² | - | O ² | R | - | R |
| KEYTO | - | - | - | O ² | - | O ² | - | - | - |
| OPTIONS | O | - | O | O | - | O | O | - | O |
| RECORD ¹⁵ | O | - | O | O | - | O | O | - | O |
| KEYSEARCH | - | - | - | - | - | - | - | - | - |
| POSITION | O ¹² | - | O ¹² | O ^{5 12} | - | O ^{5 12} | - | - | - |
| NBRKEYFLDS | - | - | - | - | - | - | - | - | - |
| INDICATORS | - | - | - | - | - | - | - | - | - |
| MODIFIED | - | - | - | - | - | - | - | - | - |
| WRITE | | | | | | | | | |
| FILE | - | R | - | - | R | - | - | R | R |
| FROM | - | R | - | - | R | - | - | R | R |
| KEYFROM | - | - | - | - | O ⁶ | - | - | R ⁶ | R ⁶ |
| OPTIONS | - | O | - | - | O | - | - | O | O |
| RECORD ¹⁵ | - | O | - | - | O | - | - | O | O |
| INDICATORS | - | - | - | - | - | - | - | - | - |
| REWRITE | | | | | | | | | |
| FILE | - | - | R | - | - | R | - | - | R |
| FROM | - | - | R | - | - | R | - | - | R |
| KEY | - | - | - | - | - | O | - | - | R |
| OPTIONS | - | - | O | - | - | O | - | - | O |
| RECORD ¹⁵ | - | - | O | - | - | O | - | - | O |
| INDICATORS | - | - | - | - | - | - | - | - | - |
| DELETE | | | | | | | | | |
| FILE | - | - | R | - | - | R | - | - | R |
| KEY | - | - | - | - | - | O | - | - | R |
| OPTIONS | - | - | O | - | - | O | - | - | O |
| RECORD ¹⁵ | - | - | O | - | - | O | - | - | O |

Data Base Files with INDEXED organization

| | FILE ACCESS | | | | | | | | |
|------------|-----------------|---|-----------------|------------------|----------------|-----------------|--------|----------------|----------------|
| | SEQUENTIAL | | | SEQUENTIAL KEYED | | | DIRECT | | |
| | I | O | U | I | O | U | I | O | U |
| READ | | | | | | | | | |
| FILE | R | - | R | R | - | R | R | - | R |
| INTO SET | R | - | R | R | - | R | R | - | R |
| KEY | - | - | - | O ² | - | O ² | R | - | R |
| KEYTO | - | - | - | O ² | - | O ² | - | - | - |
| OPTIONS | O | - | O | O | - | O | O | - | O |
| RECORD | O | - | O | O | - | O | O | - | O |
| KEYSEARCH | - | - | - | O ⁴ | - | O ⁴ | O | - | O |
| POSITION | O | - | O | O ⁵ | - | O ⁵ | - | - | - |
| NBRKEYFLDS | O ¹⁴ | - | O ¹⁴ | O ¹⁴ | - | O ¹⁴ | O | - | O |
| INDICATORS | - | - | - | - | - | - | - | - | - |
| MODIFIED | - | - | - | - | - | - | - | - | - |
| WRITE | | | | | | | | | |
| FILE | - | R | - | - | R | - | - | R | R |
| FROM | - | R | - | - | R | - | - | R | R |
| KEYFROM | - | - | - | - | O ⁷ | - | - | R ⁷ | R ⁷ |
| OPTIONS | - | O | - | - | O | - | - | O | O |
| RECORD | - | O | - | - | O | - | - | O | O |
| INDICATORS | - | - | - | - | - | - | - | - | - |
| REWRITE | | | | | | | | | |
| FILE | - | - | R | - | - | R | - | - | R |
| FROM | - | - | R | - | - | R | - | - | R |
| KEY | - | - | - | - | - | O | - | - | R |
| OPTIONS | - | - | O | - | - | O | - | - | O |
| RECORD | - | - | O | - | - | O | - | - | O |
| INDICATORS | - | - | - | - | - | - | - | - | - |
| DELETE | | | | | | | | | |
| FILE | - | - | R | - | - | R | - | - | R |
| KEY | - | - | - | - | - | O | - | - | R |
| OPTIONS | - | - | O | - | - | O | - | - | O |
| RECORD | - | - | O | - | - | O | - | - | O |

Display Files with INTERACTIVE organization

| | SEQUENTIAL ACCESS | | |
|------------|-------------------|-----------------|-----------------|
| | I | O | U |
| READ | | | |
| FILE | R | - | R |
| INTO SET | R | - | R |
| KEY | - | - | - |
| KEYTO | - | - | - |
| OPTIONS | O | - | O |
| RECORD | O ¹¹ | - | O ¹¹ |
| KEYSEARCH | - | - | - |
| POSITION | - | - | - |
| NBRKEYFLDS | - | - | - |
| INDICATORS | O ¹ | - | O ¹ |
| MODIFIED | - | - | - |
| WRITE | | | |
| FILE | - | R | R |
| FROM | - | R | R |
| KEYFROM | - | - | - |
| OPTIONS | - | O | O |
| RECORD | - | O ¹¹ | O ¹¹ |
| INDICATORS | - | O ¹ | O ¹ |

Subfiles with INTERACTIVE organization

| | SEQUENTIAL KEYED ACCESS | | |
|------------|----------------------------|------------------|------------------|
| | I | O | U |
| READ | | | |
| FILE | - | - | R |
| INTO SET | - | - | R |
| KEY | - | - | O ² |
| KEYTO | - | - | O ² |
| OPTIONS | - | - | O |
| RECORD | - | - | O ¹¹ |
| KEYSEARCH | - | - | - |
| POSITION | - | - | - |
| NBRKEYFLDS | - | - | - |
| INDICATORS | - | - | O ¹ |
| MODIFIED | - | - | O ¹⁰ |
| WRITE | | | |
| FILE | - | R | R |
| FROM | - | R | R |
| KEYFROM | - | O ^{6 9} | O ^{6 9} |
| OPTIONS | - | O | O |
| RECORD | - | O ¹¹ | O ¹¹ |
| INDICATORS | - | O ¹ | O ¹ |
| REWRITE | | | |
| FILE | - | - | R |
| FROM | - | - | R |
| KEYFROM | - | - | O |
| OPTIONS | - | - | O |
| RECORD | - | - | O ¹¹ |
| INDICATORS | - | - | O ¹ |

Display Files with CONSECUTIVE organization

| | FILE ACCESS | | | | | |
|------------|-------------------|-----------------|-----------------|-------------------|-----------------|------------------|
| | SEQUENTIAL | | | SEQUENTIAL KEYED | | |
| | I | O | U | I | O | U |
| READ | | | | | | |
| FILE | 8 | - | 8 | 8 | | 8 |
| INTO SET | R | - | R | R | - | R |
| KEY | - | - | - | - | - | - |
| KEYTO | - | - | - | - | - | - |
| OPTIONS | O | - | O | O | - | O |
| RECORD | O ¹¹ | - | O ¹¹ | O ¹¹ | - | O ¹¹ |
| KEYSEARCH | - | - | - | - | - | - |
| POSITION | O ³ | - | O ³ | O ^{3 5} | - | O ^{3 5} |
| NBRKEYFLDS | - | - | - | - | - | - |
| INDICATORS | O ^{1 13} | - | - | O ^{1 13} | - | - |
| MODIFIED | - | - | - | - | - | - |
| WRITE | | | | | | |
| FILE | - | R | - | - | R | - |
| FROM | - | R | - | - | R | - |
| KEYFROM | - | - | - | - | - | - |
| OPTIONS | - | O | - | - | O | - |
| RECORD | - | O ¹¹ | - | - | O ¹¹ | - |
| INDICATORS | - | O ¹ | - | - | - | - |

Note: Use CONSECUTIVE organization only for simple input and output. If the user is entering input and you are supplying prompt screens, or if you are providing output but allowing the user to direct program processing, for example by entering record keys, you must use INTERACTIVE organization.

Inline Files with CONSECUTIVE organization

| | SEQUENTIAL ACCESS | | |
|------------|-------------------|---|---|
| | I | O | U |
| READ | | | |
| FILE | R | - | - |
| INTO SET | R | - | - |
| KEY | - | - | - |
| KEYTO | - | - | - |
| OPTIONS | O | - | - |
| RECORD | - | - | - |
| KEYSEARCH | - | - | - |
| POSITION | O ³ | - | - |
| NBRKEYFLDS | - | - | - |
| INDICATORS | - | - | - |
| MODIFIED | - | - | - |

Printer Files with CONSECUTIVE organization

| | SEQUENTIAL ACCESS | | |
|------------|-------------------|-----------------|---|
| | I | O | U |
| WRITE | | | |
| FILE | - | R | - |
| FROM | - | R | - |
| KEYFROM | - | - | - |
| OPTIONS | - | O | - |
| RECORD | - | O ¹¹ | - |
| INDICATORS | - | O ¹ | - |

Tape and Diskette Files with CONSECUTIVE organization

| | SEQUENTIAL ACCESS | | |
|------------|-------------------|---|---|
| | I | O | U |
| READ | | | |
| FILE | R | - | - |
| INTO SET | R | - | - |
| KEY | - | - | - |
| KEYTO | - | - | - |
| OPTIONS | O | - | - |
| RECORD | - | - | - |
| KEYSEARCH | - | - | - |
| POSITION | O ³ | - | - |
| NBRKEYFLDS | - | - | - |
| INDICATORS | - | - | - |
| MODIFIED | - | - | - |
| WRITE | | | |
| FILE | - | R | - |
| FROM | - | R | - |
| KEYFROM | - | - | - |
| OPTIONS | - | O | - |
| RECORD | - | - | - |
| INDICATORS | - | - | - |

Communications and BSC Files with INTERACTIVE and CONSECUTIVE Organization

| | SEQUENTIAL ACCESS | | | | | |
|------------|-------------------|----------------|----------------|----------------|----------------|----------------|
| | CONSECUTIVE | | | INTERACTIVE | | |
| | I | O | U | I | O | U |
| READ | | | | | | |
| FILE | R | - | R | R | - | R |
| INTO SET | R | - | R | R | - | R |
| KEY | - | - | - | - | - | - |
| KEYTO | - | - | - | - | - | - |
| OPTIONS | O | - | O | O | - | O |
| RECORD | O | - | O | O | - | O |
| KEYSEARCH | - | - | - | - | - | - |
| POSITION | O ³ | - | O ³ | - | - | - |
| NBRKEYFLDS | - | - | - | - | - | - |
| INDICATORS | O ¹ | - | O ¹ | O ¹ | - | O ¹ |
| MODIFIED | - | - | - | - | - | - |
| WRITE | | | | | | |
| FILE | - | R | - | - | R | R |
| FROM | - | R | - | - | R | R |
| KEYFROM | - | - | - | - | - | - |
| OPTIONS | - | O | - | - | O | O |
| RECORD | - | O | - | - | O | O |
| INDICATORS | - | O ¹ | - | - | O ¹ | O ¹ |

Footnotes:

1. The file must contain external record definitions and must have the DDS INDARA keyword specified. The ENVIRONMENT option NOINDARA must not be specified.
2. The KEY and KEYTO options are mutually exclusive.
3. Only POSITION option NEXT allowed.
4. KEYSEARCH is not allowed if KEY is not specified.
5. POSITION may not be specified with KEY.
6. The KEYFROM(*) option is not allowed.
7. KEYFROM(expression) may only be used if the ENVIRONMENT options KEYDISP and KEYLENGTH are specified. If the ENVIRONMENT option DESCRIBED is specified, you must specify KEYFROM(*).
8. Must have the DDS INZRCD keyword specified.
9. KEYFROM may not be specified when writing to a subfile control record format.
10. MODIFIED may not be specified with KEY.
11. The RECORD option must be used with files that contain external record definitions.
12. POSITION options NXTUNQ, PRVUNQ, NXTEQL, and PRVEQL are not allowed.
13. INDICATORS may not be specified with POSITION.
14. NBRKEYFLDS is not allowed if POSITION (NEXT|PREVIOUS|FIRST|LAST) is specified.
15. The file must contain external record definitions (DDS described).

Appendix D. Conditions and Condition Codes

Conditions

This section presents conditions in alphabetical order. In general, the following information is given for each condition:

- *Description:* A discussion of the condition, including the circumstances under which the condition may be raised.
- *Implicit action:* The action taken when a condition is raised and no on-unit is currently established for that condition. In most cases, a message is issued.
- *Normal return:* The point to which control returns if the on-unit ends normally; that is, when it is not left by a GO TO statement or ended by a STOP statement. If a condition other than ERROR has been raised by the SIGNAL statement, the normal return is always to the statement immediately following SIGNAL.

The condition codes that correspond to the different situations in which a condition is raised are the values returned by the ONCODE built-in function. The condition codes for each condition, their meanings, and any OS/400 or PL/I messages associated with the condition follow this list.

If a message is issued for a condition, it will be directed to the program message queue of the run-unit's initial PL/I procedure. In addition to the message text, the message will contain the condition code, the program statement number, and the name of the external procedure containing the statement that raised the condition. In certain cases, the message will also contain the name of the file that caused the condition, the value of the key, or other information pertinent to the condition raised.

The following conditions, shown in uppercase, may be specified in an ON or SIGNAL statement:

| | |
|---------|---------------|
| ENDFILE | KEY |
| ENDPAGE | TRANSMIT |
| ERROR | UNDEFINEDFILE |

The following conditions, shown in lowercase, cannot be specified in an ON or SIGNAL statement:

CONDITIONS

| | |
|---------------|------------|
| Conversion | Storage |
| Fixedoverflow | Stringsize |
| Overflow | Underflow |
| Record | Zerodivide |

All the conditions are described in the following sections in alphabetical order.

Conversion Condition

This condition cannot be written in an ON or SIGNAL statement.

Description: The conversion condition is raised when an invalid conversion is attempted.

Implicit Action: A message is issued and the ERROR condition is raised.

Normal Return: The ERROR on-unit must not return normally.

ENDFILE Condition

Description: The ENDFILE condition is raised during a GET or READ operation by an attempt to read past the end of a file.

In record data transmission, ENDFILE is raised whenever the end of a file is found while processing a READ statement.

In stream data transmission, ENDFILE is raised while processing a GET statement if the end of a file is found before any items in the GET statement's data list have been transmitted, between transmission of two of the data items, or within a SKIP option. If the end of a file is found within a data item, or while an X-format item is being executed, the ERROR condition is raised.

If the file is not closed after ENDFILE has been raised, any subsequent GET statement, or READ statement in the same direction for that file will again raise the ENDFILE condition.

Implicit Action: A message is issued and the ERROR condition is raised.

Normal Return: Processing continues with the statement immediately following the GET or READ statement that raised the ENDFILE condition.

ENDPAGE Condition

Description: The ENDPAGE condition is raised for stream files when the last line has been printed on a page or if a LINE option or format item specifies a line number lower than the current line number. For record files, a TRANSMIT condition is raised. You can specify the number of lines for a page in the PAGESIZE option in an OPEN statement. If you do not specify PAGESIZE, a default limit of 60 is applied. The attempt to exceed the limit may be made during data transmission (including associated format items) or by the LINE or SKIP option.

ENDPAGE is raised only once for each page unless it is raised by the SIGNAL statement.

When ENDPAGE is raised, the current line number is 1 greater than the page size; it is therefore possible to continue writing on the same page. The on-unit may start a new page by processing a PAGE option or a PAGE format item, which sets the current line to 1.

If the on-unit does not start a new page, the current line number may increase indefinitely. Any subsequent LINE option or LINE format item starts a new page and sets the current line number to 1 without raising ENDPAGE, unless the current line number is equal to the specified line number and the file is positioned on column 1 of the line. In this case, the LINE specification is ignored, ENDPAGE is not raised, and no new page is started.

If ENDPAGE is raised during data transmission, the remaining data is written on the current line on return from the on-unit; the line number may have been changed by the on-unit.

If ENDPAGE results from a LINE or SKIP option or format item, then, on return from the on-unit, the action specified by LINE or SKIP is ignored.

Implicit Action: A new page is started and processing of the PUT statement continues.

Normal Return: Processing of the PUT statement continues.

ERROR Condition

Description: The ERROR condition is raised by:

- The implicit action for a condition for which that action is to issue an error message and raise the ERROR condition, such as the implicit action for the UNDEFINEDFILE condition.
- An error for which there is no other condition. This error may be a hardware detected error, such as an address exception, or a software detected error, such as processing an input/output statement for a file while another input/output statement is processing for the same file.

Implicit Action: The program ends abnormally. If the ERROR condition was raised because of an error for which no other condition exists (second case above), a message is issued before ending.

Normal Return: The program ends abnormally.

Fixedoverflow Condition

This condition cannot be written in an ON or SIGNAL statement.

CONDITIONS

Description: The fixedoverflow condition is raised when the number of digits of the intermediate result of a fixed-point arithmetic operation exceeds the maximum number of digits allowed by the implementation. The maximum for intermediate results is at least 15 for decimal fixed-point values and at least 31 for binary fixed-point values.

The fixedoverflow condition is also raised when high-order significant digits are lost in an attempted assignment to a variable or in an input/output operation. This loss may result from a conversion involving different data types, different bases, different scales, or different precisions.

Implicit Action: A message is issued and the ERROR condition is raised.

Normal Return: The ERROR on-unit must not return normally.

KEY Condition

Description: The KEY condition can be raised only during operations on keyed records. For the possible causes of the KEY condition, see Figure D-1 on page D-6.

Implicit Action: A message is issued and the ERROR condition is raised.

Normal Return: Control passes to the statement immediately following the statement that caused KEY to be raised.

Overflow Condition

This condition cannot be written in an ON or SIGNAL statement.

Description: The overflow condition is raised when the magnitude of a floating-point number exceeds the permitted maximum. No value is stored in the target.

Implicit Action: A message is issued and the ERROR condition is raised.

Normal Return: The ERROR on-unit must not return normally.

Record Condition

This condition cannot be written in an ON or SIGNAL statement.

Description: The record input/output condition can be raised only during a READ, WRITE, or REWRITE operation. For the possible causes of the condition, see Figure D-1 on page D-6.

Implicit Action: A message is issued and the ERROR condition is raised.

Normal Return: The ERROR on-unit must not return normally.

Storage Condition

This condition cannot be written in an ON or SIGNAL statement.

Description: The storage condition is raised when an ALLOCATE statement is processed without enough storage available for the request.

Implicit Action: A message is issued and the ERROR condition is raised.

Normal return: The ERROR on-unit must not return normally.

Stringsize Condition

This condition cannot be written in an ON or SIGNAL statement.

Description: The stringsize condition is raised when a string is assigned to a shorter target. If the condition is detected by a PL/I run-time routine and you specified GENOPT(*DIAGNOSE) in the CRTPLIPGM command when you compiled the program, an informational message is issued.

Implicit Action: The string is truncated to the right and processing continues.

Full Language Extension

TRANSMIT Condition

Description: The TRANSMIT input/output condition can be raised during any input or output operation in which the record is not transmitted.

Implicit Action: A message is issued and the ERROR condition is raised.

Normal Return: Processing continues with the statement immediately following the input or output statement that raised the condition.

End of Full Language Extension

UNDEFINEDFILE Condition

Description: The UNDEFINEDFILE condition is raised by an unsuccessful attempt to open a file.

If UNDEFINEDFILE is raised by an implicit opening in an input/output statement, then, upon normal return from the on-unit, processing continues with the rest of the input/output statement, provided that the file has been opened in the on-unit. If the file has not been opened, the ERROR condition is raised.

Implicit Action: A message is issued and the ERROR condition is raised.

CONDITION CODES

Normal Return: Upon the normal completion of the on-unit, control passes to the statement immediately following the statement that raised the condition. (See "Description" for the action in the case of an implicit opening.)

Underflow Condition

This condition cannot be written in an ON or SIGNAL statement.

Description: The underflow condition is raised when the magnitude of a nonzero floating-point number is smaller than the permitted minimum. (UNDERFLOW is not raised when equal numbers are subtracted.)

Implicit Action: A message is issued and processing continues with a value of zero.

Zerodivide Condition

This condition cannot be written in an ON or SIGNAL statement.

Description: The zerodivide condition is raised when an attempt is made to divide by zero.

Implicit Action: A message is issued and the ERROR condition is raised.

Normal Return: The ERROR on-unit must not return normally.

Condition Codes

The following table is a listing of the conditions that can be raised during the processing of a program. Each entry contains the condition code, an explanation of the error which raised the condition, the condition's origin, and the PL/I message issued. Conditions that can be specified are shown in uppercase; those that cannot be specified are shown in lowercase.

CONDITION CODES

| Condition and codes | Meaning | Origin | PL/I Message |
|---------------------|---|---------------------|--------------|
| Conversion 604 | Error while processing an F-format item for a GET statement: an exponent was found in the character value. | PL/I | PLI0504 |
| 612 | Error while converting from character to arithmetic. | PL/I | PLI0512 |
| 615 | Error while converting from character to bit. | PL/I | PLI0515 |
| 629 | Error while converting from picture to coded arithmetic: the character value of the picture does not correspond to its specification. | PL/I | PLI0529 |
| ENDFILE 70 | A SIGNAL ENDFILE statement was processed. | | PLI3200 |
| 71 | An end of file was detected. | CPF5001, CPF5025 | PLI3201 |
| ENDPAGE 90 | A SIGNAL ENDPAGE statement was processed. | | PLI3300 |
| 91 | An end of page was detected. | PL/I | PLI3301 |

CONDITION CODES

| Condition and codes | Meaning | Origin | PL/I Message |
|---------------------|---|---------------------|--------------|
| ERROR 9 | A SIGNAL ERROR statement was processed. | | PLI0200 |
| 1004 | Attempt to use SKIP < 0 for a print file or ≤ 0 for non-print file. | PL/I | PLI1704 |
| 1007 | A REWRITE or a DELETE statement was not preceded by a READ statement on a sequential file. Attempt to process a DELETE for a sequentially accessed file under commitment control after a commit or rollback is done. (A commit or a rollback releases all record locks.) This is caused by a logic error in the user program. The record is not deleted. Attempt to process a REWRITE for a sequentially accessed file under commitment control after a commit or rollback is done. (A commit or a rollback releases all record locks.) This is caused by a logic error in the user program. The record is not deleted. | CPF5011, CPF5147 | PLI1707 |
| 1009 | An input/output statement specifies an operation or an option that conflicts with the file. | PL/I | PLI1709 |
| 1010 | A REWRITE or DELETE was attempted using an invalid record format. | PL/I | PLI1710 |

CONDITION CODES

| Condition and codes | Meaning | Origin | PL/I Message |
|---------------------------------|--|--|----------------|
| <p>ERROR</p> <p>1011</p> | <p>Data management has detected an input/output error. Use PLIRCVMSG for more information unless the originating message was a STATUS type message.</p> | <p>CPF4500, CPF4600, CPF5100, CPF5200, CPF5300, CPF5500, CPF5600</p> | <p>PLI1711</p> |
| <p>1016</p> | <p>After the UNDEFINEDFILE condition was raised as a result of an unsuccessful attempt to implicitly open a file, the file was found unopened on normal return from the on-unit.</p> | <p>PL/I</p> | <p>PLI1716</p> |
| <p>1018</p> | <p>End of file was found while processing a data item or X-format item.</p> | <p>PL/I</p> | <p>PLI1718</p> |
| <p>1201</p> | <p>Argument passed to built-in function is invalid (for example, out of range).</p> | <p>MCH5003</p> | <p>PLI1801</p> |
| <p>3809</p> | <p>The *DIAGNOSE option was selected when the program was compiled. Therefore MI string constraintment was specified when the program was created. A violation has occurred and MCH0603 exception was signalled.</p> | <p>MCH0603</p> | <p>PLI0909</p> |
| <p>4051</p> | <p>Attempt to free a variable that has no valid allocation.</p> | <p>PL/I</p> | <p>PLI2151</p> |

CONDITION CODES

| Condition and codes | Meaning | Origin | PL/I Message |
|---------------------|---|--|--------------|
| ERROR 4059 | When an ALLOCATE or FREE statement was being processed, an element of the free area chain was found that did not contain valid information. | PL/I | PLI2159 |
| 5000 | The number of arguments passed to a procedure did not match the number of parameters expected. | MCH0801 | PLI2200 |
| 8090 | A floating-point operation had invalid operands. | MCH1209 | PLI1900 |
| 8095 | Address exception. | MCH3601 | PLI1905 |
| 8096 | Alignment error. | MCH0602 | PLI1906 |
| 9000 | Function check or other serious error during processing. | CPF9999 or any of several MCH messages | PLI2000 |
| 9002 | Attempt to process a GO TO statement which references an invalid statement label. | PL/I | PLI2002 |

CONDITION CODES

| Condition and codes | Meaning | Origin | PL/I Message |
|----------------------|---|------------------|--------------|
| Fixedoverflow 310 | <p>The fixedoverflow condition is raised when the number of digits of the intermediate result of a fixed-point arithmetic operation exceeds the maximum number of digits allowed by the implementation. The maximum for intermediate results is at least 15 for decimal fixed-point values and at least 31 for binary fixed-point values.</p> <p>The fixedoverflow condition is also raised when high-order significant digits are lost in an I/O operation. This loss may result from a conversion involving different data types, different bases, different scales, or different precisions.</p> | MCH1210, PL/I | PLI0600 |

CONDITION CODES

| Condition and codes | Meaning | Origin | PL/I Message |
|---------------------|---|---------------------|--------------|
| KEY | | | |
| 50 | A SIGNAL KEY statement was processed. | | PLI3400 |
| 51 | The specified key cannot be found. | CPF5006, CPF5020 | PLI3401 |
| 52 | When duplicate keys are not allowed, an attempt was made to add a keyed record that has the same key as a record already in the file. | CPF5008, CPF5026 | PLI3402 |
| | A WRITE was done to an existing relative record number. | PL/I | |
| 53 | When duplicate keys are not allowed, an attempt was made to add a keyed record that has the same key as a record already in another file over which there is a common unique keyed index. | CPF5034 | PLI3403 |
| | A WRITE or REWRITE was done to an existing relative record number. | PL/I | |
| 54 | A key conversion was attempted on a CONSECUTIVE or INTERACTIVE file. The key is less than or equal to 0. | PL/I | PLI3404 |
| 55 | Key specification is null string. | PL/I | PLI3405 |
| 59 | An invalid key was detected by OS/400. The error is described by PLIRCVMSG. | CPF5090 | PLI3409 |
| 62 | READ statement with the NXTEQL or PRVEQL POSITION option was used and there was no next or previous equal key in the access path. | CPF5006 | PLI3412 |

CONDITION CODES

| Condition and codes | Meaning | Origin | PL/I Message |
|----------------------------------|---|-------------------------------------|--|
| Overflow 300 | A floating-point number is greater than the maximum size permitted. | MCH1206 | PLI0700 |
| Record 21 22 23 | <p>The record variable is smaller than the record size; in a READ INTO statement the rest of the record is lost.</p> <p>The record variable is larger than the record size; in a READ INTO statement the rest of the variable is undefined.</p> <p>The length of the record variable in a WRITE or REWRITE statement is zero or the variable is too short to contain the embedded key. No transmission occurs. This condition is raised only for data base files or device files that do not contain DDS.</p> | <p>PL/I</p> <p>PL/I</p> <p>PL/I</p> | <p>PLI3601</p> <p>PLI3602</p> <p>PLI3603</p> |
| Storage 8085 | An ALLOCATE statement requested more storage than is available to the program. | MCH2804, MCH5401 | PLI0485 |
| Stringsize none | The stringsize condition is raised when a string is assigned to a shorter target. | PL/I | PLI1000 |

CONDITION CODES

| Condition and codes | Meaning | Origin | PL/I Message |
|---------------------|---|---------------------------------|--------------|
| TRANSMIT | | | |
| 40 | A SIGNAL TRANSMIT statement was processed. | | PLI3700 |
| 43 | File input/output error. Use PLIRCVMSG for more information, unless the originating message was a STATUS type message. | CPF4700, CPF4800, CPF5000 | PLI3703 |
| 44 | Printer overflow line detected. | CPF5004 | PLI3704 |
| 47 | Record locked. Use PLIRCVMSG for more information, unless the originating message was a STATUS type message. The requested record is currently locked. Note: Attempt to process a read for a record in a file which is opened for UPDATE. The record is locked, and remains locked until the next file operation is processed. The program should ensure that any such locks held by the program are released. | CPF5027, CPF5032 | PLI3707 |
| 48 | Data conversion through input/output with a logical file. | CPF5029, CPF5035 | PLI3708 |
| 49 | File size limit exceeded. | CPF5018, CPF5043 | PLI3709 |

CONDITION CODES

| Condition and codes | Meaning | Origin | PL/I Message |
|---------------------|--|---------------------------------|--------------|
| UNDEFINEDFILE | | | |
| 80 | A SIGNAL UNDEFINEDFILE statement was processed. | | PLI3800 |
| 81 | The attributes in a DECLARE statement conflict with those of an explicit or implicit OPEN. | PL/I | PLI3801 |
| 82 | The file attributes conflict with the physical characteristics of the AS/400 file, such as a conflict between the file organization and the device type. | PL/I | PLI3802 |
| 84 | File or member not found. | CPF4101, CPF4102 | PLI3804 |
| 86 | The value of the LINESIZE option on the OPEN statement is too large for the logical record length of the device. | PL/I | PLI3806 |
| 89 | Authorization failure. | CPF4104, CPF4236 | PLI3809 |
| 93 | An error was detected by OS/400 while opening a file. Use PLIRCVMSG to obtain more information, unless the originating message was a STATUS type message. A file under commitment control has been opened, but either the STRCMTCTL command has not been issued or this file is not journaled to the same file as the other files under commitment control. The file is not opened. | CPF4100, CPF4200, CPF4300 | PLI3813 |

CONDITION CODES

| Condition and codes | Meaning | Origin | PL/I Message |
|-------------------------|--|---------------------|--------------|
| UNDEFINEDFILE (cont) | | | |
| 96 | Invalid TITLE option. | PL/I CPF4128 | PLI3816 |
| 97 | Unable to allocate objects for file. | | PLI3817 |
| 98 | Attributes of the currently open file do not match the attributes specified in the OPEN statement. | PL/I | PLI3818 |
| Underflow none | A floating-point number is smaller than the minimum size permitted. | MCH1207 | PLI1200 |
| Zerodivide 320 | A divide operation was attempted using zero as the divisor. | MCH1211, MCH1214 | PLI1300 |

Appendix E. EBCDIC CODES

The EBCDIC codes may be represented by different characters on different terminals or printers.

| Character | EBCDIC Code | Character | EBCDIC Code |
|-----------|-------------|-----------|-------------|
| blank | 40 | s | A2 |
| . | 4B | t | A3 |
| < | 4C | u | A4 |
| (| 4D | v | A5 |
| + | 4E | w | A6 |
| | 4F | x | A7 |
| & | 50 | y | A8 |
| \$ | 5B | z | A9 |
| * | 5C | A | C1 |
|) | 5D | B | C2 |
| ; | 5E | C | C3 |
| ┘ | 5F | D | C4 |
| - | 60 | E | C5 |
| / | 61 | F | C6 |
| , | 6B | G | C7 |
| % | 6C | H | C8 |
| - | 6D | I | C9 |
| > | 6E | J | D1 |
| : | 7A | K | D2 |
| # | 7B | L | D3 |
| @ | 7C | M | D4 |
| ' | 7D | N | D5 |
| = | 7E | O | D6 |
| a | 81 | P | D7 |
| b | 82 | Q | D8 |
| c | 83 | R | D9 |
| d | 84 | S | E2 |
| e | 85 | T | E3 |
| f | 86 | U | E4 |
| g | 87 | V | E5 |
| h | 88 | W | E6 |
| i | 89 | X | E7 |
| j | 91 | Y | E8 |
| k | 92 | Z | E9 |
| l | 93 | 0 | F0 |
| m | 94 | 1 | F1 |
| n | 95 | 2 | F2 |
| o | 96 | 3 | F3 |
| p | 97 | 4 | F4 |
| q | 98 | 5 | F5 |
| r | 99 | 6 | F6 |
| | | 7 | F7 |

| Character | EBCDIC Code |
|-----------|-------------|
| 8 | F8 |
| 9 | F9 |



Appendix F. Converting from System/38 to the AS/400 System

Your Choice of Two Environments: AS/400 System or the System/38 Environment

The AS/400 System offers many enhancements over System/38. However, because a great many PL/I programs have been written for the System/38 computer, and because many programmers are already familiar with the System/38, the AS/400 System also supports these programs.

The CL command CALL QCL changes the AS/400 System screen display to appear to the user as a System/38. This is known as the System/38 Environment. When you are in this environment, you can enter and compile PL/I programs, and do anything else, as if you were using a System/38 machine. To exit from the System/38 Environment, you would enter RETURN. To use the System/38 Environment, you must have the files QCL and QCMD in your library list.

Compiling in the System/38 Environment

You use the CL command CRTPLIPGM to compile PL/I source programs in the System/38 Environment just as in the AS/400 System environment. CRTPLIPGM is used in a similar manner in the two environments with a few exceptions. They are as follows:

For the PGM parameter the following option applies:

*CURLIB: If a library name is not specified, the program is stored in *CURLIB. The program must not already exist in the library

For the SRCFILE parameter the following option applies:

*LIBL: The library list is used to find the source file.

For the INCFILE parameter the following options applies:

*LIBL: The library list is used to find the source file.

For the PRTFILE parameter the following option applies:

*LIBL: The library list is used to find the print file.

The PUBAUT parameter is used instead of the AUT parameter.

For the PUBAUT parameter the following options apply:

*NORMAL: The program is treated as *CHANGE in the AS/400 System environment.

*ALL: The public has complete authority for the program.

TWO ENVIRONMENTS

*NONE: The program is treated as *EXCLUDE in the AS/400 System environment.

Examples

The following command compiles a program named PAYROLL.

```
CRTPLIPGM PAYROLL TEXT('Payroll Program')
```

The source program is in the default source file QPLISRC, in a member named PAYROLL. A compiler listing is generated. The program runs under the user's user profile, and can be run by any system user.

The following command creates a PL/I program named PARTS.

```
CRTPLIPGM PGM(PARTS) +  
  SRCFILE(PARTDATA.MYLIB) +  
  OPTION(*XREF *OPT) PUBAUT(*NONE) +  
  TEXT('This program displays all parts data')
```

The program object is stored in the library QGPL. The source program is in the PARTS member of the source file PARTDATA in the library MYLIB. A compiler listing, cross-reference listing, and compiler-option list is generated. This program, which cannot be used by the public, can be run by the owner or another user that the owner has explicitly authorized by name in the CL command GRTOBJAUT (Grant Object Authority).

Writing Programs in the System/38 Environment

The following %INCLUDE directive and the TITLE parameter of the OPEN statement show the difference in the filename when using PL/I in the System/38 environment.

Using the %INCLUDE Directive

file-name

An identifier of up to ten characters. The file is located by using the *LIBL search list in effect at compile time. The file name cannot begin with a numeric and cannot contain periods; the possible characters are A-Z, 0-9, #, @, _ . You cannot name your file SYSLIB.

member-name

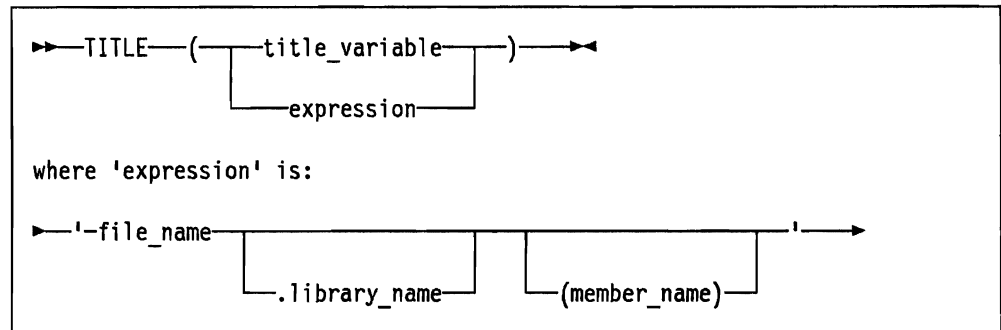
An identifier of up to ten characters. The name must be unique within one file. The name cannot contain a period or start with a numeric character.

Using the %INCLUDE Directive for External File Descriptions

file-name

An identifier of up to 10 characters. The file is located by using the *LIBL search list in effect at compile time. The file name cannot begin with a numeric and cannot contain periods; the possible characters are A-Z, 0-9, #, @, _ . You cannot name your file SYSLIB.

Syntax of TITLE Parameter of the OPEN Statement



title-variable

May be any valid program variable whose value is a valid System/38 expression.
For example:

```

ACCOUNTS_RECEIVABLE
      = 'ACCOUNTS.ACCTLIB(ACCOUNT1)';
OPEN FILE (ACCTS) UPDATE
      TITLE (ACCOUNTS_RECEIVABLE);

```

expression

Must consist of valid System/38 file, library and member name.

If library-name is omitted, *LIBL is assumed.

If member-name is omitted, the first member in the file is used.

TWO ENVIRONMENTS




Appendix G. Glossary of Abbreviations

| Abbreviation | Stands For | Definition |
|--------------|---|---|
| ANSI | American National Standards Institute | An organization sponsored by the Computer and Business Equipment Manufacturers Association for establishing voluntary industry standards. |
| ASCII | American National Standard Code for Information | The standard code used for information interchange between data processing systems, data communications systems, and associated equipment. The code uses a coded character set consisting of 7-bit coded characters (8 bits including parity check). The set consists of control characters and graphic characters. |
| BSC | Binary Synchronous Communication | A form of telecommunication line control that uses a standard set of transmission control characters and control character sequences, for binary synchronous transmission of binary-coded data between stations. |
| CL | Control Language | The set of all commands with which a user requests functions. |
| OS/400 | Operating System/400 | The system support licensed program for the AS/400 System. It provides many functions that are fully integrated in the system such as work management, data base data management, job control, message handling, security, programming aids, and service. |
| DDM | Distributed Data Management | A program product that allows an application program or user on a source system to access data files on remote systems connected by a communications network that also uses DDM. |
| DDS | Data Description Specifications | A description of the user's data base or device files that is entered into the system using a fixed-form syntax. The description is then used to create files. |

| Abbreviation | Stands For | Definition |
|---------------------|--|---|
| EBCDIC | Extended Binary-Code Decimal Inter-change Code | A coded character set consisting of 8-bit coded characters. |
| F | Function Key | A keyboard key that is used to request a specific system function. |
| FCFC | First Character Forms Control | A method for controlling the format of printed output. The first character of each record determines the format. |
| HLL | High-Level Language | A programming language that relieves the programmer from the rigors of machine level or assembler level programming; for example, RPG III, CL, BASIC, and COBOL. |
| K. | Kilobyte | The primary unit of measure for storage capacity; 1 K = 1024 bytes. |
| NaN | Not-A-Number | In binary floating-point concepts, a value, not interpreted as mathematical value, which contains a mask state and a sequence of binary digits. |
| PL/1 | Programming Language One | A programming language designed for numeric scientific computations, business data processing, systems programming and other applications. |
| QGPL | General-Purpose Library | The library provided by the Control Program Facility to contain user-oriented, IBM-provided objects and user-created objects not explicitly placed in a different library when they are created. |
| SDLC | Synchronous Data Link Control | A discipline conforming to subsets of the Advanced Data Communication Control Procedures (ADDCCP) of the American National Standards (ANS) and High-level Data Link Control (HDLC) of the International Organization for Standardization, for managing synchronous, code-transparent, serial-by-bit information transfer over a link connection. Transmission exchanges may be duplex or half-duplex over switched or non-switched links. The configuration of the link connection may be point-to-point, multipoint or loop. |

| Abbreviation | Stands For | Definition |
|---------------------|---------------------------|---|
| SEU | Source Entry Utility | The part of the Utilities Program Product used by the operator to enter and update source and procedure members. |
| SQL | Structured Query Language | A relational data base management system which allows data access in both interactive and noninteractive systems. |




Glossary of Terms

additive attribute. A file description attribute that must be stated explicitly or implied by another explicitly stated attribute. Contrast with *alternative attribute*.

allocated variable. A variable to which storage has been assigned.

alternative attribute. A file description attribute that is chosen from a group of attributes. Contrast with *additive attribute*.


arithmetic comparison. A comparison of signed numeric values. See also *bit comparison*, *character comparison*.



array. A collection of one or more elements with identical attributes, grouped into one or more dimensions.


array of structures. An array whose elements are structures that have identical names, levels, and element attributes.

array variable. A variable that represents an aggregate of data items that must have identical attributes. Contrast with *structure variable*.



assignment statement. A statement that gives a value to a variable. It always contains the assignment symbol (=).

based variable. A variable that provides attributes for data (such as data located in a buffer) for which the storage address is provided by a pointer. It does not identify a fixed location in storage.




begin-block. A block that is activated by error-handling on-conditions or through the normal flow of control.

binary fixed-point value. An integer consisting of binary digits and having an optional binary point. Contrast with *decimal fixed-point value*.

binary floating-point value. An approximation of a real number in the form of a significand, which can be considered as a binary fraction, and an exponent, which can be considered as an integer exponent to the base of 2. Contrast with *decimal floating-point value*.

bit comparison. A left-to-right, bit-by-bit comparison of binary digits. See also *arithmetic comparison*, *character comparison*.



bit constant. Either a series of binary digits enclosed in apostrophes and followed immediately by B or B1, or a

series of hexadecimal digits enclosed in apostrophes and followed immediately by B4. Contrast with *character constant*.

bit value. A sequence of binary digits stored in consecutive bits.

block. A sequence of statements, processed as a unit, that specifies the scope of names and the allocation of storage for names declared within it. Contrast with *do group*.

break character. The underscore symbol (`_`). It can be used to improve the readability of identifiers. For instance, a variable could be called `OLD_INVENTORY_TOTAL` instead of `OLDINVENTORYTOTAL`.

built-in function. A predefined function, such as a commonly used arithmetic function or a function necessary to language facilities (for example, a function for manipulating strings or converting data). It is called by a built-in function reference.

built-in function reference. A built-in function name, having an optional and possibly empty argument list, that represents the value returned by the built-in function.

character comparison. A left-to-right, character-by-character comparison according to the collating sequence. See also *arithmetic comparison*, *bit comparison*.

character constant. A sequence of characters enclosed in apostrophes; for example, `'CONSTANT'`.

coded arithmetic data. Data items that represent numeric values and are characterized by their base (decimal or binary), scale (fixed-point or floating-point), and precision (the number of digits each can have).

combined nesting depth. The sum of all `PROCEDURE/BEGIN/ON, DO, SELECT, and IF...THEN...ELSE` nestings in the program.

comparison operator. An operator that can be used in an arithmetic, string, or logical relation to indicate the comparison to be done between the terms in the relation. The comparison operators are = (equal to), > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to), != (not equal to), >= (not greater than), <= (not less than).

composite symbol. A symbol that consists of more than one special character; for example, <=, **, ->, and /*.

condition. An exceptional situation, either an error (such as an overflow), or an expected situation (such as the end of an input file). When a condition is raised (detected), the action established for it is processed. See also *established action* and *implicit action*.

connected aggregate. An array or structure that has no inherited dimensions.

control variable. A variable that is used to control the running of a program, as in a DO statement.

DDM file. A AS/400 file that is associated with a remote file that is accessed using DDM. The DDM file provides the information needed for a local (source) system to locate a remote (target) system and to access the file at the target system where the requested data is stored.

data aggregate. A group of data items that can be referred to either individually or collectively. There are two types of aggregates: arrays and structures.

data list. In PL/I stream data transmission, a list of the data items used in GET EDIT and PUT EDIT statements. Contrast with *format list*.

decimal fixed-point constant. A constant consisting of one or more decimal digits with an optional decimal point.

decimal fixed-point value. A rational number consisting of a sequence of decimal digits with an assumed position of the decimal point. Contrast with *binary fixed-point value*.

decimal floating-point constant. A value made up of a significand that consists of a decimal fixed-point constant, and an exponent that consists of the letter E followed by an optionally signed integer constant not exceeding three digits.

decimal floating-point value. An approximation of a real number, in the form of a significand, which can be considered as a decimal fraction, and an exponent, which can be considered as an integer exponent to the base of 10. Contrast with *binary floating-point value*.

default. Is used to describe a value, attribute, or option that is assumed when none has been specified.

dimension attribute. An attribute that specifies the number of dimensions of an array and indicates the bounds of each dimension.

directive. A statement that directs the operation of the compiler.

do group. A sequence of statements, processed as a unit, that may be a non-iterative do group (processed once or possibly not at all) or an iterative do group (processed several times, once, or not at all). Contrast with *block*.

entry constant. The label prefix of a PROCEDURE statement.

entry data item. A data item that represents an entry point to a procedure.

entry reference. An entry constant, an entry variable reference, or a function reference that returns an entry value.

entry variable. A variable to which an entry value can be assigned.

established action. The action taken when a condition is raised. See also *implicit action* and *ON-statement action*.

explicit declaration. A DECLARE statement that specifies the attributes of a name. A procedure's name is declared by the PROCEDURE statement: the statement's label is declared as the name of the procedure. Contrast with *implicit declaration*.

expression. A representation of a value; it can consist of constants, variables, and function references, along with operators or parentheses or both.

extent. The number of integers between and including the lower and upper bounds of an array.

extralingual character. Any EBCDIC code that is not an alphabetic character, a special character, or a digit.

file constant. A name declared for a file and for which a complete set of file description attributes exists during the time that the file is open, and with which each file must be associated.

file description attribute. A keyword that describes the characteristics of a file. See also *alternative attribute* and *additive attribute*.

format list. In PL/I stream data transmission, a list specifying the format of the data item on the external medium. Contrast with *data list*.

function. A procedure that has a RETURNS option in the PROCEDURE statement. A function ends by processing a RETURNS (expression) statement and returning a scalar value to the point of calling. Contrast with *subroutine*.

function reference. An entry constant or an entry variable, either of which must represent a function, followed by a possibly empty argument list. Contrast with *subroutine reference*.

identifier. A single alphabetic character or a string of alphabetic characters, digits, and break characters that starts with an alphabetic character.

implicit. Is used to describe the action taken in the absence of an explicit statement.

implicit action. The action established for a condition when the program is activated and that remains established unless overridden by the processing of an ON statement for the same condition. Contrast with *ON-statement action*.

inherited dimensions. For a structure field, those dimensions that are inherited from the containing structures. If the structure field is a scalar variable, the dimensions consist entirely of its inherited dimensions. If the structure field is an array, its dimensions consist of its inherited dimensions plus its explicitly declared dimensions. A structure field with one or more inherited dimensions is referred to as an unconnected aggregate. Contrast with *connected aggregate*.

initial procedure. An external procedure, called by a calling program, that activates a PL/I program.

instruction pointer. A pointer that provides addressability for an MI instruction in a program.

integral boundary. The multiple of any 8-bit unit of information on which data can be aligned.

internal procedure. A procedure that is contained in another block. Contrast with *external procedure*.

keyword. An identifier that when used in a defined context takes on a specific meaning, such as an action taken or the attributes of data.

keyword statement. A simple statement that begins with a keyword indicating the function of the statement.

label. An identifier that names a statement so that it can be referred to at some other point in the program. Sometimes called a label prefix.

label constant. A name written as the label prefix of any statement other than PROCEDURE. Contrast with *label variable*.

label prefix. See *label*.

label value. An attribute that identifies a statement in the running program.

label variable. An identifier that contains the label of a statement so that the label can be referred to at some other point in the program. Contrast with *label constant*.

language character. Any one of the alphabetic characters, the digits 0 through 9, and twenty special characters.

level-number. A number that precedes a name in a DECLARE statement and specifies the organization of the structure in that statement.

name. Any identifier that the user assigns to a variable or to a constant. Sometimes called a user-defined name.

null statement. A statement that contains only the semicolon symbol (;).

null string. A character or bit string with a length of zero.

ON-statement action. The action explicitly established for a condition when the condition is raised. The ON-statement action overrides or suspends any previously established action unless it is overridden by a further ON-statement for the same condition or until the block it was processed in ends. Contrast with *implicit action*.

operational expression. An expression that consists of one or more operations.

picture data. Arithmetic data represented in character form.

picture specification. A data item that has a numeric value but that can also be represented as a character value according to the editing characters specified in the item's declaration.

pointer. A type of variable that identifies a location in storage.

pointer value. A value that identifies the location of data in storage.

precision. The number of digits contained in a fixed-point data item, or the minimum number of significant digits (excluding the exponent) maintained for a floating-point data item.

problem data. Coded arithmetic, bit, character, and picture data which represents values processed by the program. Contrast with *program control data*.

procedure. A block that can be activated from various points within a program by use of a CALL statement

and can process data passed to it from the block in which it was called. See also *external procedure* and *internal procedure*.

procedure calling level. The calling level that is incremented when an internal procedure is called recursively. The procedure calling level cannot be specified on AS/400 debug commands, and only the last (most recent) procedure calling level is available for debugging. Contrast with *program calling level*.

program control data. Pointer, label, entry, and file data that is used to control the processing of a PL/I program. Contrast with *problem data*.

program calling level. The calling level incremented when a program or external procedure is called recursively. The program calling level can be specified on AS/400 debug commands through the INVLVL parameter. Contrast with *procedure calling level*.

record data transmission. The transmission of data in the form of separate records. Contrast with *stream data transmission*.

recursive procedure. An active procedure that can be called from within itself or from within another active procedure.

run unit. A set of PL/I programs, each of which is called by some other PL/I program within the set, except for the initially called program, which is called from outside the set. A PL/I run unit is suspended when a program in the run unit calls a non-PL/I program, and is resumed when the called program returns control to the PL/I program that called it. A PL/I run unit is ended when the initially called PL/I program returns control to the non-PL/I program that originally called the initial program and so started the run unit.

scalar. A type of program object that contains either string or numeric data. It provides the byte string it is mapped to with representation and operational characteristics. Contrast with *pointer*.

scalar variable. A variable that represents a single data item.

scope. The part of the program in which a data item can be referenced.

statement. A grouping of identifiers, constants, and delimiters that makes up do groups and blocks. The end of a statement is indicated by a semicolon (;). See also *keyword statement*, *assignment statement*, and *null statement*.

stream data transmission. The transmission of data in which the organization of the data into records is ignored and the data is treated as though it were a continuous stream of individual data values in character form. Contrast with *record data transmission*.

string. (1) A series of things, such as characters, in a line. (2) In PL/I, a contiguous sequence of characters or bits that is treated as a single data item. (3) A group of auxiliary storage devices connected to a system. The order and location in which each device is connected to the system determines the physical address of the device.

structure. A collection of data items that need not have identical attributes. Contrast with *array*.

structure variable. A variable that represents an aggregate of data items that might not have identical attributes. Contrast with *array variable* and *scalar variable*.

subroutine. A procedure that has no RETURNS option in the PROCEDURE statement. Contrast with *function*.

subroutine call. An entry reference that must represent a subroutine, followed by an optional and possibly empty argument list that appears in a CALL statement. Contrast with *function reference*.

undefined. Is used to indicate something that is not defined by the language and that may change without notice. Thus, programs that seem to work correctly when referencing undefined results do so by chance and are in error.

Index

A

A-format item 11-31, 11-32

abbreviations

AFT for AFTER (value) 7-17
ALLOC for ALLOCATE (statement) 5-22
ASM for ASSEMBLER (option) 12-36
AUTO for AUTOMATIC (attribute) 12-41
BFR for BEFORE (value) 7-17
BIN for BINARY (attribute) 12-10
BIN for BINARY (built-in function) 15-8
CHAR for CHARACTER (attribute) 12-16
CHAR for CHARACTER (built-in function) 15-8
COL for COLUMN (format item) 11-34
DCL for DECLARE (statement) 7-1, 12-1
DEC for DECIMAL (attribute) 12-10
DEC for DECIMAL (built-in function) 15-9
DIM for DIMENSION (built-in function) 15-10
ENV for ENVIRONMENT (attribute) 7-1, 12-8
EQL for EQUAL (value) 7-17
EXT for EXTERNAL (attribute) 4-16
FOFL for fixedoverflow (condition) D-3, D-10
INIT for INITIAL (attribute) 12-42
INT for INTERNAL (attribute) 4-16
NXT for NEXT (value) 7-18
OFL for overflow (condition) D-4, D-12
OTHER for OTHERWISE (statement) 13-18
PIC for PICTURE (attribute) 12-19
PROC for PROCEDURE (statement) 14-2
PRV for PREVIOUS (value) 7-18
PTR for POINTER (attribute) 12-30
SEQL for SEQUENTIAL (attribute) 12-7
STG for STORAGE (built-in function) 15-22
UFL for underflow (condition) D-6, D-16
UNAL for UNALIGNED (attribute) 5-7, 12-40
UNDF for UNDEFINEDFILE (condition) 10-1
VAR for VARIABLE (attribute) 12-37
ZDIV for zerodivide (condition) D-6, D-16

abnormal program ending 2-22

ABS built-in function 15-5

access

direct 12-7, 11-9

sequential 12-7, 11-9

access paths of AS/400 data base files 6-4

arrival sequence 6-5

examples 8-7, 8-14–8-22

keyed sequence 6-5

examples 8-5, 8-10

obligatory coding of INDEXED option 7-2

accessing PL/I on the AS/400 System. 1-1

ACOS built-in function 15-6

additive attributes 12-6, 12-8

ENVIRONMENT 12-8

KEYED 12-7

PRINT 12-7

ADDR built-in function 12-31, 15-6

ADDR 15-6

AFT value

See AFTER value

AFTER value 7-17

See also KEYSEARCH parameter of the
OPTIONS option

aggregate arguments 15-5

ALIAS keyword of DDS: effect on declarations

generated by %INCLUDE 8-78

ALIGNED attribute 12-40, 5-7

alignment 5-9

attributes

ALIGNED 5-7, 12-40

defaults 5-8, 12-40

UNALIGNED 5-8, 12-40

boundary

of data 5-7

ALLOC (ALLOCATE) statement 5-22

ALLOCATE statement (abbr: ALLOC) 5-22

allocation of storage

automatic 5-16

static 5-18

alphabetic characters B-14

alternative attributes 11-9, 12-6

defaults 12-6

DIRECT 11-9

alternative attributes (*continued*)

- INPUT 11-10, 11-23
- OUTPUT 11-10, 11-23
- RECORD 11-3
- SEQUENTIAL 11-9
- STREAM 11-3
- UPDATE 11-10, 11-23
- ANSI standard, deviations B-1, B-13
- apostrophe (') B-14
- argument 9-1, 14-1
 - and parameter, association of 14-9
 - dummy 14-10
 - in CALL statement 14-7
- arithmetic 9-5, 5-26, 9-5
 - built-in functions 15-2, 15-3
 - ABS 15-5
 - BINARY 15-8
 - conversion of picture arguments 15-3
 - converted precision 15-3
 - DECIMAL 15-9
 - DIVIDE 15-10
 - FIXED 15-11
 - FLOAT 15-11
 - MAX 15-13
 - MIN 15-13
 - MOD 15-14
 - ROUND 15-20
 - SIGN 15-21
 - TRUNC 15-25
 - comparison 9-10, 9-13
 - data assignment 5-27
 - padding 5-26
 - precision and accuracy 5-27
 - undefined result, fixed-point 5-26
 - operations 9-5, 9-9
 - results 9-5
 - operators 4-4
 - overflow 9-6
 - target 5-29
- array 12-38
 - assignment 13-1
 - bounds 12-38, 12-39
 - bounds for automatic variables 5-19
 - element 5-1
 - extent 12-39
 - INITIAL attribute 12-43
 - of structures 5-5, 5-6
 - reference 9-1

array (*continued*)

- subscripted reference 5-3
- subscripts 5-2
- variable 5-1
- array handling built-in functions 15-3
 - arguments 15-3
 - DIMENSION 15-10
 - HBOUND 15-11
 - LBOUND 15-12
 - result 15-3
- array mapping 5-11, 5-12
- arrays 5-1, 5-3, 12-38
- arrays of structures 5-5, 5-6
 - declaration 5-5
 - inheritance of dimensions 5-6
 - subscripted qualified array reference 5-6
- arrival sequence access path of AS/400 data base files 6-5
 - program examples 8-7, 8-14–8-22
- ASA forms control characters 7-4
- ASIN built-in function 15-6
- ASM option
 - See ASSEMBLER option
- ASSEMBLER option 12-36
- assignment 13-1, 5-24, 5-27
 - aggregate 13-1
 - arithmetic data 5-26
 - by initialization of variables 5-24
 - BY NAME 13-3
 - by passing a dummy argument 14-10
 - by returning a value 5-24
 - in input/output operations 5-24
 - problem data 13-1
 - program control data 13-1
 - scalar 13-1
 - statement 4-2, 13-2
 - examples 13-2
 - string data 5-25
 - structure 13-1
 - symbol (=) 4-4, 9-10
 - to control variable 13-7
 - to UNSPEC pseudovvariable 15-25
- association of arguments and parameters 14-9, 14-11
- ATAN built-in function 15-7
- ATAND built-in function 15-7

ATANH built-in function 15-7
 attributes 12-1, 12-8
 additive 12-6
 ALIGNED 5-7, 12-40
 alternative 12-6
 defaults 12-6
 AUTOMATIC 5-18
 BASED 5-19, 12-42
 BINARY 12-10
 BIT 12-16
 BUILTIN 12-36
 CHARACTER 12-16
 DECIMAL 12-10
 description 12-1
 DIRECT 11-9
 ENVIRONMENT 7-1, 7-10
 EXTERNAL 4-16, 7-11, 12-40
 FILE 12-7
 FIXED 12-10
 FLOAT 12-10
 INITIAL 5-17
 INPUT 11-10, 11-23
 INTERNAL 4-16, 7-11, 12-40
 KEYED 12-7
 LABEL 12-31
 of returned value 14-4
 OPTIONS 12-36
 OUTPUT 11-10, 11-23
 parameter 14-3
 POINTER 12-30
 precision 12-10
 PRINT 12-7
 RECORD 11-3
 required in declarations 4-13
 RETURNS 12-35
 SEQUENTIAL 11-9
 STATIC 12-42
 STREAM 11-3
 UNALIGNED 5-8, 12-40
 UPDATE 11-10, 11-23
 VARIABLE 12-37, 12-38
 VARYING 12-18
 authority, file 6-3
 AUTO attribute
 See AUTOMATIC attribute
 automatic 5-18
 array specification 5-19
 storage allocation 5-18

automatic (*continued*)
 storage and attribute 5-19
 storage class 5-16
 variable 5-18
 AUTOMATIC attribute 5-18

B

B insertion character 12-23
 B or B1 bit constant identifier 12-16
 B-format item 11-32
 B-, B1-, and B4- format items 11-32, 11-34
 base 12-10
 attributes
 BINARY 12-10
 DECIMAL 12-10
 default 12-10
 coded arithmetic data 12-10
 based 5-20, 5-23
 storage and attribute 5-19
 ADDR built-in function 12-31
 NULL built-in function 12-31
 storage class 5-16
 variable
 reference 5-20
 variable reference and pointer
 qualification 5-20, 5-21
 variables and input/output 5-24
 BASED attribute 5-19, 12-42
 basic program structure 4-1
 basic reference 9-1
 BEFORE value 7-17
 See also KEYSEARCH parameter of the
 OPTIONS option
 BEGIN statement 4-11
 pairing with END statement 13-10
 begin-block activation 4-11
 begin-block ending 4-12
 begin-blocks 4-11, 4-6, 4-11–4-12
 as on-units 10-3
 BFR value
 See BEFORE value
 BIN attribute
 See BINARY attribute
 BINARY and DECIMAL attributes 12-10
 BINARY attribute 12-10
 BINARY built-in function 15-8

- binary fixed-point data 12-12–12-13
- binary fixed-point value 12-12
- binary floating-point data 12-14
- binary floating-point value 12-14
- binary synchronous communications files 6-10
- bit 9-9, 12-16
 - B or B1 12-16
 - B4 12-16
 - comparison 9-11
 - constant 12-16
 - conversion
 - to arithmetic 5-31
 - to character 5-33
 - data 12-16, 12-17
 - maximum length 12-17
 - operations 9-10
 - operators 9-9, 4-4
 - target 5-33
 - value 12-16
 - null 12-17
- BIT and CHARACTER attributes 12-16
- BIT attribute 12-16
- BIT built-in function 15-8
- bit data 12-16
 - alignment 5-8, 12-40
 - defaults 5-8, 12-40
- bit format items
 - B
 - See B-format item
 - B1
 - See B1-format item
 - B4
 - See B4-format item
- blanks 4-5
- block activation 4-7
- block ending 4-7, 4-9
- BLOCK option of the ENVIRONMENT
 - attribute 7-7
- blocks 4-6–4-12
 - activation 4-7
 - begin 4-6
 - ending 4-7
 - external 4-6
 - internal 4-6
 - nested 4-12
 - procedure 4-6
- boundary
 - alignment 5-9
 - byte 5-7
 - doubleword 5-7
 - halfword 5-7
 - integral 5-7
 - quadword 5-7
 - word 5-7
- bounds of an array 12-38
- breakpoints 3-5
 - example 3-5
- browsing compiler listings 2-4
- BSC files
 - See binary synchronous communications files
- BUFSIZE option of the ENVIRONMENT attribute 7-5
 - buffer length default 7-5
- built-in function reference 15-1
- built-in functions 15-1
 - ABS 15-5
 - ACOS 15-6
 - ADDR 15-6
 - aggregate arguments 15-5
 - arithmetic 15-2
 - array handling 15-3
 - ASIN 15-6
 - ATAN 15-7
 - ATAND 15-7
 - ATANH 15-7
 - BINARY 15-8
 - BIT 15-8
 - calling 15-1
 - CHARACTER 15-8
 - classification 15-2
 - computational 15-2
 - condition handling 15-3
 - conversion 5-28
 - conversion of arguments 15-2
 - COPY 15-8
 - COS 15-9
 - COSD 15-9
 - COSH 15-9
 - DATE 15-9
 - DECIMAL 15-9
 - declaration 15-1
 - DIMENSION 15-10
 - DIVIDE 15-10
 - empty argument lists 15-5

built-in functions (*continued*)

EXP 15-11
FIXED 15-11
FLOAT 15-11
HBOUND 15-11
INDEX 15-12
input/output 15-4
LBOUND 15-12
LENGTH 15-12
LINENO 15-12
LOG 15-12
LOG10 15-13
LOG2 15-13
mathematical 15-3
MAX 15-13
MIN 15-13
miscellaneous 15-4
MOD 15-14
NULL 15-14
ONCODE 15-14
ONFILE 15-15
ONKEY 15-15
PLIRETV 15-19
PLISHUTDN 15-20
reference 15-1
ROUND 15-20
SAMEKEY 15-21
SIGN 15-21
SIN 15-22
SIND 15-22
SINH 15-22
SQRT 15-22
storage control 15-4
string handling 15-2
SUBSTR 15-23
TAN 15-24
TAND 15-24
TANH 15-24
TIME 15-24
TRANSLATE 15-24
TRUNC 15-25
UNSPEC 15-25
used in conversion 5-28
VERIFY 15-26
built-in functions and pseudovariables 15-1, 15-26
built-in names 15-5

built-in subroutines 15-4, 15-5
PLICOMMIT 8-59, 15-4
PLIDUMP 3-14, 15-4
PLIIOFDB 3-17, 15-5
PLIOPNFDB 3-17, 15-5
PLIRCVMMSG 15-5
PLIRETC 15-4
PLIROLLBACK 8-60, 15-5
BUILTIN attribute 12-36–12-37
BY NAME assignment 13-3, 13-5
by-name-parts-list 13-3
rules for assignment 13-3
BY option 13-7
byte boundary 5-7
B1-format item 11-32
B4 bit constant identifier 12-16
B4-format item 11-32

C

calculating string length and precision 5-29
CALL statement 14-7, 14-9
calling a block 4-10
calling a non-PL/I program 2-23
calling a PL/I program from a non-PL/I program 2-24
calling a procedure 14-4, 14-11
calling levels 3-10
ceil values
table of 9-9
CHAR attribute
See CHARACTER attribute
character 12-18
comparison 9-11
constant
maximum length 12-18
conversion
to arithmetic 5-31
to bit 5-34
data 12-18
target 5-31
value
null 12-18
of picture data 12-19
variable declaration 12-18
CHARACTER attribute 12-16
CHARACTER built-in function 15-8

character format item
 See A-format item
 characters B-14
 and EBCDIC codes B-15
 extralingual B-15
 language
 alphabetic B-14
 digits B-14
 special B-14
 CL commands
 ADDFM 8-2
 ADDTRC 3-7
 BGNCMTCTL 7-7, 8-58, 8-62
 CALL 2-22
 CHGDBG 3-3
 CHGHLLPTR 3-12
 CRTLF 8-2
 CRTPLIPGM 2-5, 7-12
 CRTTAPF 7-6
 CRTxxxF 7-6
 DSPTRCDTA 3-7
 ENDCMTCTL 7-7, 8-58
 ENTDBG 3-3
 GRTOBJAUT 2-15
 MONMSG 2-22
 OVRDBF 6-2, 6-7, 7-6
 OVRTAPF 7-6
 RCLRSC 7-11
 RTVJOBA 2-22
 RVKOBJAUT 6-3
 SBMJOB 6-10
 WRKJOB 2-22
 classification of built-in functions 15-2, 15-4
 CLOSE statement 7-11, 11-8
 following run time error 7-13
 closing an unopen file 11-8
 coded arithmetic data
 base 12-10
 binary fixed-point 12-12
 binary floating-point 12-14
 conversion to bit 5-33
 conversion to character 5-31
 decimal fixed-point 12-11
 decimal floating-point 12-13
 precision 12-10
 scale 12-10
 COL format item
 See COLUMN format item
 colon (:) 4-5
 COLUMN format item 11-34
 combination of operations 9-14, 9-16
 COMMA in %INCLUDE directive 8-75
 comma (,) 4-5
 comments 4-5
 commercial "at" sign (@) B-14
 commitment control 8-58
 performance considerations 8-61
 PLICOMMIT 8-59
 PLIROLLBACK 8-60
 program examples 8-64–8-73
 specified by ENVIRONMENT attribute 7-6
 using record locks 8-62
 COMMITTABLE option of the ENVIRONMENT attribute 7-6, 8-59
 communication with non-PL/I programs 2-23
 communications files 6-10
 comparison 9-10
 operations 9-13
 problem data 9-11
 program control data 9-11
 tables 9-12
 operators 9-10
 compiler directives
 See also directives
 entering 2-4
 in source programs 2-5
 using 2-16
 *PROCESS 2-16
 %INCLUDE 2-16
 %PAGE 2-18
 %PROCESS 2-16, 2-18
 %SKIP 2-19
 compiler output 2-19
 compiler segments A-5
 compiler service A-1
 compiling source programs 2-5
 CRTPLIPGM command 2-5
 composite symbols B-15
 computational built-in functions 15-2
 concatenating copies of a string
 See COPY built-in function
 concatenation 9-14
 operations 9-14
 operator (||) 9-14

- condition codes D-1
 - conversion condition D-7
 - ENDFILE condition D-7
 - ENDPAGE condition D-7
 - fixedoverflow condition D-11
 - KEY condition D-12
 - overflow condition D-13
 - record condition D-13
 - UNDEFINEDFILE condition D-15
 - underflow condition D-16
 - zerodivide condition D-16
- condition handling 10-1, 15-3
 - built-in functions 15-4
 - ONCODE 15-14
 - ONFILE 15-15
 - ONKEY 15-15
 - scope of result 15-3
 - scope of values 10-4
- conditional picture characters 12-23
- conditions D-1, 10-2, D-1–D-6, D-16
 - action when raised 10-1
 - conversion D-2
 - ENDFILE D-2
 - ENDPAGE D-7, D-2
 - ERROR D-7, D-3
 - established action 10-2
 - fixedoverflow D-3, D-10
 - groups of 10-1
 - in ON statement 10-2
 - in SIGNAL statement 10-4
 - investigating cause of 15-3
 - KEY D-11, D-4
 - list of D-6, D-16
 - overflow D-4, D-12
 - record D-4, D-13
 - storage D-5, D-13
 - stringsize D-5, D-13
 - TRANSMIT D-5, D-13
 - UNDEFINEDFILE D-14, D-5
 - underflow D-6, D-16
 - unspecifiable 10-2
 - zerodivide D-6, D-16
- conditions in ON and SIGNAL statements 10-1, 10-2
 - ENDFILE D-2, 10-1
 - ENDPAGE D-7, D-2, 10-1
 - ERROR D-7, D-3, 10-1
 - KEY D-11, D-4, 10-1
- conditions in ON and SIGNAL statements (*continued*)
 - UNDEFINEDFILE D-14, D-5, 10-1
- CONSECUTIVE option of the ENVIRONMENT
 - attribute 7-2
- constants 4-2
 - bit 12-16
 - character 12-18
 - decimal fixed-point 12-11
 - decimal floating-point 12-13
 - entry 12-32
 - file 11-3
 - label 12-31
 - named 12-9
- contained in 4-12
- contextual declaration 4-13
 - built-in functions or built-in subroutines 4-13, 15-1
 - overriding 4-13
 - scope 4-13
- contextual declaration of built-in functions or built-in subroutines 15-1
- control characters 7-4
- control format items 11-31, 11-29
 - COLUMN format item 11-34
 - LINE format item 11-39
 - PAGE format item 11-39
 - processing 11-29
 - SKIP format item 11-39
 - X-format item 11-40
- control language (CL) 1-1
- conversion
 - to arithmetic 5-29
 - to bit 5-33
 - to character 5-31
- conversion built-in functions 5-28, 5-29
- conversion condition D-2
- conversion of problem data
 - See data conversion
- conversion rules 5-29, 5-34
- converted precision 15-3
- COPY built-in function 15-8
- copying a string
 - See COPY built-in function
- COS built-in function 15-9
- COSD built-in function 15-9

- COSH built-in function 15-9
- CR picture character 12-26
- Create PL/I Program command
 - See CRTPLIPGM command
- creating null pointer value 15-4
- creating programs 2-1
 - See also compiling source programs
- credit and debit characters 12-26, 12-30
 - CR (credit) picture character pair 12-26
 - DB (debit) picture character pair 12-26
- CRTPLIPGM command 2-5
 - compiler output 2-5
 - examples 2-15
 - first CRTPLIPGM command screen 2-6
 - options and parameters 2-15
 - second CRTPLIPGM command screen 2-9
 - third CRTPLIPGM command screen 2-12
 - *DIAGNOSE option of the GENOPT parameter 2-16
- CTLASA option of the ENVIRONMENT attribute 7-4
- currency symbol (\$) B-14

D

- data
 - alignment 5-7
 - conversion 5-24, 5-27, 9-4, 9-5, 9-11
 - format items 11-31, 11-29
 - lists 11-23
 - mapping 5-9
 - organization 5-1
 - specifications 11-23
 - transmission statements 11-2, 11-10, 11-23
 - transmitted 11-10
- data aggregate 5-1, 12-38
 - arguments 15-5
 - arrays 5-1, 12-38
 - assignment 13-1
 - structures 5-3
- data alignment 5-7
 - bit 5-8, 12-40
 - byte 5-7
 - halfword 5-7
 - quadword 5-7
 - word 5-7
- data attributes
 - ALIGNED 5-7, 12-40
 - BINARY 12-10

- data attributes (*continued*)
 - BIT 12-16
 - BUILTIN 12-36
 - CHARACTER 12-16
 - DECIMAL 12-10
 - dimension 12-38
 - FILE 12-7
 - FIXED 12-10
 - FLOAT 12-10
 - label 12-31
 - POINTER 12-30
 - precision 12-10
 - structure 5-3
 - UNALIGNED 5-8, 12-40
 - VARIABLE 12-37, 12-38
 - VARYING 12-18
- data attributes of target 5-27
- data base files 6-4, 6-8, 8-1
 - access paths 6-4
 - externally described 8-2
 - logical files 6-4
 - members of 6-6
 - physical files 6-4
 - program examples 8-2—8-22
 - program-described 8-2
 - record formats 6-5
 - use of 8-1
- data conversion 5-27, 5-24, 5-27—5-35, 9-5
 - accuracy of values 5-31
 - built-in functions
 - BINARY 15-8, 5-28
 - BIT 15-8, 5-28
 - CHARACTER 15-8, 5-28
 - DECIMAL 15-9, 5-28
 - FIXED 15-11, 5-28
 - FLOAT 15-11, 5-28
 - by assignment 5-29
 - examples 5-34
 - in arithmetic operations
 - exponentiation 9-5
 - non-exponentiation 9-5
 - in comparison operations 9-11
 - in operational expressions 9-4
 - on assignment
 - precision 5-26
 - string length 5-25
 - operands 5-28
 - problem data 5-27

data conversion (*continued*)

- program control data 5-27
- rules 5-29
- to arithmetic
 - from arithmetic 5-29
 - from bit 5-31
 - from character 5-31
- to bit
 - from arithmetic 5-33
 - from character 5-34
 - from fixed-point binary 5-33
 - from fixed-point decimal 5-33
 - from floating-point binary 5-34
 - from floating-point decimal 5-34
 - from picture 5-33
- to character
 - from bit 5-33
 - from coded arithmetic 5-31
 - from fixed-point binary 5-31
 - from fixed-point decimal 5-32
 - from floating-point binary 5-32
 - from floating-point decimal 5-32
 - from picture 5-33
- data conversion and assignment 5-24, 5-27
- data conversion in arithmetic operations 9-5
- data description specifications
 - examples
 - DDS for subfiles 8-29
 - display file DDS 8-23
 - logical file DDS 8-4
 - physical file DDS 8-3
 - printer file DDS 8-50
 - %INCLUDE examples 8-79
 - mapping DDS data types to a PL/I program 8-77
 - record format definition 6-5
 - use of 8-2, 8-23, 8-29, 8-50
 - used with externally described data files 6-11
- data format items 11-31, 11-29, 11-34, 11-37
 - A-format item 11-31
 - association with data item 11-29
 - B-format item 11-32
 - B1-format item 11-32
 - B4-format item 11-32
 - E-format item
- data lists 11-27, 11-23, 11-27—11-28
 - array data transmission 11-41
 - assignment 11-41

data lists (*continued*)

- data list item 11-27
- iterative specification 11-28
- structure data transmission 11-41
- data management
 - record formats 6-11
 - externally described 6-11
 - program-described 6-12
 - system considerations
 - device independence 6-1
 - file independence 6-1
 - security 6-3
 - system override considerations 6-2
 - types of files 6-4—6-10
 - data base files 6-4, 8-1
 - device files 6-8, 8-49
- data mapping 5-9, 5-15
 - DDS to PL/I 8-77
- data organization 5-1, 5-6
- data set organization 11-9
- data sets 11-1, 11-2—11-9
 - recording data in 11-9
- data specifications 11-23
- data transmission
 - input 11-1
 - output 11-1
 - record 11-1
 - statements 11-2, 11-10, 11-23
 - stream 11-1
- data transmission statements 11-1, 11-10, 11-23
 - DELETE 11-16
 - GET 11-24
 - PUT 11-25
 - READ 11-11
 - record 11-1
 - REWRITE 11-15
 - stream 11-23
 - WRITE 11-14
- data transmitted 11-10
- DATE built-in function 15-9
- DB picture character 12-26
- DCL statement
 - See DECLARE statement
- DDM files 6-10
- DDS
 - See data description specifications

- DDS to PL/I mapping 8-77
- debug
 - active blocks in a program 3-13
 - calling levels 3-10
 - changing varying length strings 3-12
 - displaying level numbers 3-13
 - floating-point variables 3-12
 - fully qualified names 3-11
 - PLIDUMP 3-14
 - PL/I pointers 3-12
 - PL/I storage 3-10
 - references to static variables 3-13
 - scoping of names 3-11
 - specifying variables by ODV number 3-13
 - using 3-10
- debugging aids 3-1
 - CPF test and debug features
 - breakpoints 3-5
 - test libraries 3-3
 - traces 3-7
 - PL/I debugging features
 - ON conditions 3-17
 - PLIDUMP 3-14, 15-16
 - PLIIOFDB 3-17, 15-16
 - PLIOPNFDB 3-17, 15-17
- DEC attribute
 - See DECIMAL attribute
- DECIMAL attribute 12-10
- DECIMAL built-in function 15-9
- decimal fixed-point constant 12-11
 - precision 12-11
- decimal fixed-point data 12-11, 12-12
- decimal fixed-point value 12-11
- decimal floating-point constant 12-13
 - precision 12-13
- decimal floating-point data 12-13
- decimal floating-point value 12-13
- declaration
 - CAUTION for multiple declarations 4-14
 - contextual 4-13
 - explicit 4-13
- DECLARE statement 7-1, 12-1
- declaring 12-31
 - arrays 12-38
 - arrays of structures 5-5
 - BASED variables 5-19, 12-42
 - binary fixed-point variables 12-12
 - binary floating-point variables 12-14
- declaring (*continued*)
 - bit variables 12-17
 - built-in functions or built-in subroutines 15-1
 - character variables 12-18
 - decimal fixed-point variables 12-12
 - decimal floating-point variables 12-13
 - entry constant 12-32
 - entry variables 12-32
 - file constants 12-7
 - label constants 12-31
 - label variables 12-31
 - parameters 14-2
 - pointer variables 12-30
 - structures 5-4, 12-38–12-39
- declaring a built-in function or built-in subroutine 15-1
- defaults 12-41
 - A-format item output field-width 11-31
 - alternative attributes 12-6
 - bit variable length 12-16
 - buffer length 7-5
 - B, B1, and B4 format items output field-width 11-33
 - character variable length 12-16
 - do-group expression_3 13-7
 - E-format item fractional-digits 11-35
 - F-format item 11-37
 - for SKIP option 11-26
 - line size 11-5
 - lower bound 12-39
 - page size 11-6
 - precision 12-11
 - scope 14-3
 - SKIP format item relative-line 11-40
 - %SKIP statement number-of-lines 2-19
- defining a procedure 14-1, 14-4
- DELETE statement 11-16
 - OPTIONS option 7-13
 - with RECORD parameter 7-17
- delimiters 4-3, 4-5
 - assignment symbol (=) 4-5
 - blank 4-5
 - colon (:) 4-5
 - comma (,) 4-5
 - comment 4-5
 - operators 4-4
 - parentheses () 4-5
 - percent statements 4-5

- delimiters (*continued*)
 - period (.) 4-5
 - pointer (->) 4-5
 - semicolon (;) 4-5
- DESCRIBED option of the ENVIRONMENT
 - attribute 7-6
- descriptions of built-in functions and
 - pseudovariabes 15-5, 15-26
- device files 6-8, 6-10, 8-49
 - BSC files 6-10
 - communications files 6-10
 - DDM files 6-10
 - display files 6-8, 8-22
 - externally described 8-50
 - inline files 6-10
 - program example 8-50
 - program-described 8-50
 - use of 8-49
- device independence 6-1
 - spooling 6-2
- digit and decimal point characters 12-22, 12-23
 - alignment of value 12-23
- digits B-14
- DIM built-in function
 - See DIMENSION built-in function
- dimension attribute 12-2, 12-38
- DIMENSION built-in function 15-10
- DIRECT attribute 11-9, 12-7
- directives 2-16, 2-22, 4-1—4-2
 - definition 4-1
 - %INCLUDE 8-73, 8-82
 - %PAGE 2-18
 - %PROCESS 2-18
 - %SKIP 2-19
- diskette device files
 - obligatory use of BUFSIZE option 7-5
- display files 6-8, 6-9, 8-22
 - externally described 8-23
 - program examples 8-23—8-49
 - subfiles 6-9
 - use of 8-22
- displaying and printing messages 3-3
- DIVIDE built-in function 15-10
- DO statement 13-5, 13-10
 - examples 13-8—13-10
 - pairing with END statement 13-5, 13-10

- do-group 13-5
 - effect of processing 13-7
 - ending 13-5, 13-11
 - iterative 13-5
 - maximum nesting 13-8
 - noniterative 13-5
 - simple 13-6
 - TO and BY versus REPEAT 13-7
 - transferring into 13-8
- doubleword boundary 5-7
- drifting characters 12-24
- dummy argument 14-10
- DUMMYDCL null character string generated by
 - %INCLUDE directive 8-75
- dynamic storage allocation 5-16

E

- E-format item 11-34, 11-37
- EBCDIC codes B-15, E-1
- edit-directed data transmission
 - See stream data transmission
- editing source programs
 - description 2-1
 - entering SQL statements 2-5
 - PL/I syntax checker 2-2
 - using SEU 2-2
- editing the source program 2-1
- effect of recursion on automatic variables 14-12
- elementary expression 9-1
- elements of a PL/I statement 4-2, 4-5
 - delimiters 4-3
 - identifiers 4-3
- ELSE unit 13-13
- empty argument lists 15-5
- END statement 13-10, 13-11
 - block ending 13-11
 - do-group ending 13-11
 - in a function 13-11
 - pairing with BEGIN, DO, and PROCEDURE statements 13-11
 - program ending 13-11
- ENDFILE condition D-2, D-7
- ending
 - a program 4-6
 - begin-blocks 4-12
 - blocks 4-7
 - procedures 4-11
 - program processing 2-22

ENDPAGE condition D-2, D-3, D-7
 entering source programs 2-1
 entering SQL statements 2-5
 entry 12-32
 argument 14-7
 built-in name 14-7
 constant 14-2
 data 12-36
 reference 12-32, 14-7
 variable 12-32
 entry data 12-32
 ENV attribute
 See ENVIRONMENT attribute
 ENVIRONMENT attribute 7-1, 7-10, 12-8
 BLOCK option 7-7
 BUFSIZE option 7-5
 COMMITTABLE option 7-6, 8-59
 CTLASA option 7-4
 DESCRIBED option 7-6
 file locking options
 EXCL 7-3
 EXCLRD 7-3
 file organization options
 CONSECUTIVE 7-2
 INDEXED 7-2
 INTERACTIVE 7-3
 key options
 KEYDISP 7-4
 KEYLENGTH 7-4
 NOINDARA option 7-8
 EQL value
 See EQUAL value
 EQUAL value 7-17
 See also KEYSEARCH parameter of the
 OPTIONS option
 ERROR condition D-3, D-7—D-10
 finding the cause 10-1
 raised by DELETE or REWRITE 8-64
 raised by I/O statement options 6-3
 error dump 3-16
 error finding in programs 3-1
 established action 10-2, 10-4
 implicit 10-2
 ON statement 10-2
 scope of 10-4
 EXCL option of the ENVIRONMENT
 attribute 7-3

EXCLRD option of the ENVIRONMENT attri-
 bute 7-3
 EXP built-in function 15-11
 explicit declaration 4-13
 of a parameter 4-13
 scope of 4-13
 explicit declaration of built-in functions or built-in
 subroutines 15-1
 exponentiation 9-5
 expression 9-1
 operational 9-4
 scalar 9-1
 expressions as subscripts 5-3
 EXT attribute
 See EXTERNAL attribute
 extent 12-39
 extents for automatic variables 5-19
 EXTERNAL attribute 4-16, 7-11, 12-40
 defaults 12-41
 scope of name 4-16
 external name 11-4, 12-41
 maximum length 12-41
 external procedure 4-9, 4-6
 external text
 inclusion of 2-16
 externally described files
 data base files 8-2
 device files 8-50
 display files 8-23
 use of %INCLUDE directive 8-75
 externally described record formats 6-11
 defining with DDS 6-11
 level checking 6-12
 use of %INCLUDE directive 8-75
 extralingual characters B-15

F

F-format item 11-37, 11-39
 factoring
 attributes 12-2
 nesting 12-2
 structure level numbers 12-2
 factoring of attributes 4-13, 12-2
 field name 5-3
 file 11-4
 additive attributes 12-6
 alternative attributes 12-6
 characteristics, specified by ENVIRONMENT
 attribute 7-4

file (*continued*)
 closing 11-8, 7-11
 constant 11-3
 declaration
 ENVIRONMENT attribute 7-1
 EXTERNAL attribute 7-11
 INTERNAL attribute 7-11
 default construction for OPEN statement 11-5
 description with DDS 8-2, 8-23, 8-29, 8-50
 determining current state of 15-4
 independence 6-1
 locking 6-6, 7-3
 management 8-1—8-58
 opening 7-11
 examples 11-7
 implicit 11-6
 implied attributes 11-6
 sources of information 11-6
 organization, specified by ENVIRONMENT
 attribute 7-2
 redirection 6-3
 scope 7-11, 12-7
 security 6-3
 authority 6-3
 ownership 6-3
 sharing 7-11
 spooling 6-2
 types 6-4
 See also file types supported by AS/400 PL/I
 FILE attribute 12-7
 file locking options of the ENVIRONMENT attri-
 bute 7-3
 EXCL 7-3
 EXCLRD 7-3
 FILE option
 for record data transmission 11-18
 for stream data transmission 11-26
 in CLOSE statement 11-8
 in GET statement 11-26
 in OPEN statement 11-26
 in PUT statement 11-26
 file types supported by AS/400 PL/I 6-4
 data base files 6-4, 8-1
 logical files 6-4
 members 6-6
 physical files 6-4
 device files 6-8, 8-49
 BSC files 6-10
 communications files 6-10

file types supported by AS/400 PL/I (*continued*)
 device files (*continued*)
 DDM files 6-10
 inline files 6-10
 subfiles 6-8, 8-22
 FIXED and FLOAT attributes 12-10
 FIXED attribute 12-10
 FIXED built-in function 15-11
 fixed overflow condition D-3, D-4, D-10—D-11
 fixed-point binary to bit conversion 5-33
 fixed-point binary to character conversion 5-31
 fixed-point decimal to bit conversion 5-33
 fixed-point decimal to character conversion 5-32
 fixed-point format item
 See F-format item
 FLOAT attribute 12-10
 FLOAT built-in function 15-11
 floating-point binary to bit conversion 5-34
 floating-point binary to character conversion 5-32
 floating-point decimal to bit conversion 5-34
 floating-point decimal to character
 conversion 5-32
 floating-point format item
 See E-format item
 floating-point variables in test environment 3-12
 FOFL condition
 See fixed overflow condition
 format items
 control
 COLUMN 11-34
 LINE 11-39
 PAGE 11-39
 SKIP 11-39
 X 11-40
 data
 A 11-31
 B 11-32
 B1 11-32
 B4 11-32
 E 11-34
 F 11-37
 format lists 11-23
 format items 11-29
 iteration factor 11-28
 FREE statement 5-23
 FROM option 11-19

- fully qualified names 3-11
- function reference 14-4, 14-6
- functions 14-1
 - arithmetic built-in 15-2
 - array handling built-in 15-3
 - attributes of returned value 14-4
 - built-in 15-1
 - calling 14-1
 - condition handling built-in 15-3
 - conversion of value 14-3
 - data processed 14-1
 - definition 14-1
 - input/output built-in 15-4
 - mathematical built-in 15-3
 - miscellaneous built-in 15-4
 - return from 14-1
 - RETURN statement in 14-4
 - storage control built-in 15-4

G

- general statements 13-1, 13-20
- GET statement 11-24
- GO TO statement 13-11, 13-13
- GOTO statement
 - See GO TO statement
- group
 - See do-group

H

- halfword boundary 5-7
- HBOUND built-in function 15-11
- how to read syntax diagrams vi

I

- IBM extensions to AS/400 PL/I B-1, B-13
- identifiers 4-3
 - maximum length 4-3
- identifying location of variable 15-4
- IF statement 13-13, 13-14
 - comparison operation 9-10
 - examples 13-14
- implicit action for condition 10-2, D-1
 - conversion condition D-2
 - ENDFILE condition D-2
 - ENDPAGE condition D-3
 - ERROR condition D-3
 - fixedoverflow condition D-4

- implicit action for condition (*continued*)
 - KEY condition D-4
 - overflow condition D-4, D-12
 - record condition D-4
 - storage condition D-5, D-13
 - stringsize condition D-5
 - TRANSMIT condition D-5
 - UNDEFINEDFILE condition D-5, D-16
 - underflow condition D-6, D-16
 - zerodivide condition D-6, D-16
- implicit opening 11-6, 11-7
 - implied attributes 11-6
- including text in source program
 - See %INCLUDE directive
- independence
 - device 6-1
 - file 6-1
- INDEX built-in function 15-12
- INDEXED option of the ENVIRONMENT attribute 7-2
- indicators
 - examples 8-43—8-49
 - use with %INCLUDE directive 8-76
- INDICATORS element type in %INCLUDE directive 8-74
- INDICATORS parameter of the OPTIONS option 7-19
 - with READ 7-19
 - with REWRITE 7-19
 - with WRITE 7-19
- industry standards viii
- infix operation 9-4
- infix operator 9-1
- inherited dimensions
 - See arrays of structures
 - initial
 - See initial procedure
- INIT attribute
 - See INITIAL attribute
- INITIAL attribute 5-17, 5-18, 12-42
 - and static external variables 5-17
- INITIAL attribute and inherited dimensions 5-17
- initial procedure 4-6, 14-7
- initial value 5-17, 12-43
- initialization 5-17, 12-43
 - array variable 5-17, 12-43
 - scalar variable 5-17, 12-43

- inline data files 6-10
- inline files 6-10, 7-12
- input 11-1
- input and output 11-1, 11-8
 - statements 11-1
- input and output statements
 - See data transmission statements
- INPUT attribute 11-10, 11-23, 12-7
- INPUT element type in %INCLUDE directive 8-74
- INPUT option in OPEN statement 11-4
- input/output built-in functions 15-4
 - LINENO 15-12
 - SAMEKEY 15-21
- INPUT, OUTPUT, and UPDATE
 - attributes 11-10, 11-23, 12-7
 - default 12-7
- insertion characters 12-23, 12-24
 - and drifting string 12-25
 - B 12-23
 - . 12-23
 - / 12-23
 - , 12-23
- INT attribute
 - See INTERNAL attribute
- integral boundary 5-7
- INTERACTIVE option of the ENVIRONMENT attribute 7-3
- interlanguage calls 2-23
 - calling a PL/I program from a non-PL/I program 2-24
 - matching PL/I attributes in
 - BASIC 2-27
 - CL 2-27
 - COBOL 2-27
 - RPG 2-24
 - non-PL/I program 2-23
- INTERNAL and EXTERNAL attributes 4-16, 12-40
- internal and external procedures 4-9, 4-10
- INTERNAL attribute 4-16, 7-11, 12-40
 - defaults 12-41
 - scope of name 4-16
- internal procedure 4-9, 4-6
- internal to 4-12
- interrupting program processing 2-22

- INTO option 11-12, 11-18—11-19
- ITERATE statement 13-14, 13-15
 - examples 13-15
- iteration-factor
 - in format list 11-28
 - in INITIAL attribute 12-43
- iterative do-group 13-5

K

- KEY condition D-4, D-11—D-12
 - raised by duplicate key 8-62
- KEY element type in %INCLUDE directive 8-74
- KEY option 11-12, 11-20
- key options of the ENVIRONMENT
 - attribute 7-3
 - KEYDISP 7-4
 - KEYLENGTH 7-4
- KEYDISP option of the ENVIRONMENT attribute 7-4
- KEYED attribute 12-7
- keyed sequence access path of AS/400 data base files 6-5
 - obligatory coding of INDEXED option 7-2
 - program examples 8-5, 8-10
- KEYFROM option 11-12, 11-21
 - RESTRICTIONS 7-3, 7-6
- KEYLENGTH option of the ENVIRONMENT attribute 7-4
- KEYSEARCH parameter of the OPTIONS
 - option 7-17
 - values
 - AFTER 7-17
 - BEFORE 7-17
 - EQLAFT 7-17
 - EQLBFR 7-17
 - EQUAL 7-17
 - with READ 7-17
- KEYTO option 11-12, 11-21—11-22
- keyword 4-3
- keyword statement 4-2
- keywords
 - See under individual keywords

L

- label 4-1, 12-31
 - constant
 - declaration 12-31
 - prefix 4-2
 - definition 4-1
 - for DECLARE statement 12-1
 - reference in GO TO statements 13-12
 - statement
 - See label prefix
 - variable 12-31
 - declaration 12-31
- LABEL attribute 12-31
- label data and attribute 12-31, 12-32
- language characters 4-1, B-15
 - alphabetic characters B-14
 - composite symbols B-14
 - digits B-14
 - special characters B-14
- LBOUND built-in function 15-12
- LEAVE statement 13-15, 13-16–13-17
 - examples 13-17
- LENGTH built-in function 15-12
- level checking
 - of externally defined files 6-12
 - preventing with CL commands 7-6
 - specified by ENVIRONMENT attribute 7-6
- level number, structure 5-4
 - factoring 12-2
 - range 12-1
- level, structure 12-1
- LINE format item 11-39
- line length specification
 - See LINESIZE option
- LINE option 11-27
- LINENO built-in function 15-12
- lines on page, specifying number of
 - See PAGESIZE option in OPEN statement
- LINESIZE option 11-5
- listing control directives 2-16
 - %PAGE 2-18
 - %SKIP 2-19
- locate mode 11-11
- locking
 - files 6-6, 7-3
 - records 6-7

- LOG built-in function 15-12
- logical files 6-4
 - DDS example 8-4
- logical level, structure 5-4
- LOG10 built-in function 15-13
- LOG2 built-in function 15-13
- lower-bound of array 12-38

M

- MAIN option 14-3
- major structure name 5-3
- mapping 5-12
 - array data 5-11
 - DDS to PL/I 8-77
 - scalar data 5-10
 - structure data
 - example 5-13
 - rules 5-12
- mathematical built-in functions 15-3
 - ACOS 15-6
 - ASIN 15-6
 - ATAN 15-7
 - ATAND 15-7
 - ATANH 15-7
 - conversion of arguments 15-3
 - COS 15-9
 - COSD 15-9
 - COSH 15-9
 - EXP 15-11
 - LOG 15-12
 - LOG10 15-13
 - LOG2 15-13
 - scale of arguments 15-3
 - scale of result 15-3
 - SIN 15-22
 - SIND 15-22
 - SINH 15-22
 - SQRT 15-22
 - TAN 15-24
 - TAND 15-24
 - TANH 15-24
- MAX built-in function 15-13
- maximum
 - See also range
 - array dimensions 12-39
 - array length 12-39
 - array upper-bound 12-39
 - bit constant length 12-17

maximum (*continued*)
 bit variable length 12-16
 block nesting 4-10
 blocks in external procedure 4-6
 character constant length 12-18
 character variable length 12-16
 digits in exponent 11-37, 12-13
 do-group nesting 13-8
 external entry name length 14-2
 external name length 12-41
 fractional digits in E-format item 11-34
 fractional digits in F-format item 11-37
 IF statement nesting 13-14
 iterative specifications in data list 11-28
 label prefix 4-1
 level number in DECLARE statement 12-1
 line number in LINE format item 11-39
 line size 11-5
 maximum depth 13-14
 member names in %INCLUDE statement 2-17
 name length 4-3
 on-units concurrently active 10-3
 on-units in external procedure 10-3
 page size 11-6
 parameters in procedure 14-2
 picture specification
 digit positions 12-20
 length 12-20
 precision 12-11
 binary fixed-point 12-11
 binary floating-point 12-14
 decimal fixed-point 12-11
 decimal floating-point 12-11
 relative-line number in SKIP format item or option 11-40
 statement label prefix 4-1
 structure length 5-5, 12-39
 structure level number in DECLARE statement 5-5, 12-39
 structure nesting 5-5, 12-39
 %INCLUDE statement nesting 2-17
 %SKIP statement number-of-lines 2-19
 members of data base files 6-6
 messages
 displaying 3-3
 printing 3-3
 using 3-1

MIN built-in function 15-13
 minimum
 See also range
 bit variable length 12-16
 character variable length 12-16
 line size 11-5
 page size 11-6
 relative-line number in SKIP format item or option 11-40
 %SKIP statement number-of-lines 2-19
 minor structure name 5-3
 miscellaneous built-in functions 15-4
 DATE 15-9
 PLIRETV 15-19
 TIME 15-24
 MOD built-in function 15-14
 MODIFIED parameter of the OPTIONS
 option 7-20
 with READ 7-20
 move mode 11-11
 multiple declarations 4-14
 multiple pointer qualification 5-21, 5-22

N

name 4-3, 4-12, 9-1, 12-1
 factoring 12-2
 in an END statement 13-11
 scope 4-12
 name-list 12-1
 named constant 12-9
 NBRKEYFLDS parameter of the OPTIONS
 option 7-18
 with READ 7-18
 nested blocks 4-12
 new line, starting with OPEN statement
 See LINESIZE option
 NEXT value 7-18
 See also POSITION parameter of the OPTIONS option
 NOINDARA option of the ENVIRONMENT
 attribute 7-8
 non-PL/I routines 12-36
 noniterative do-group 13-5
 nonstructure parameter descriptor 12-33
 normal return from condition D-1
 conversion condition D-2
 ENDFILE condition D-2
 ENDPAGE condition D-3

normal return from condition (*continued*)
 ERROR condition D-3, D-7
 fixedoverflow condition D-4
 KEY condition D-4
 overflow condition D-4, D-12
 record condition D-4
 TRANSMIT condition D-5
 UNDEFINEDFILE condition D-6
 zerodivide condition D-6, D-16
 null bit constant 12-17
 NULL built-in function 15-14, 12-31
 NULL 15-14
 null character constant 12-18
 null statement 4-2, 13-17–13-18
 examples 13-18
 number of digits
 picture specification 12-21
 precision attribute 12-11
 number sign (#) B-14
 numeric value of picture data 12-19
 NXT value
 See POSITION parameter of the OPTIONS
 option

O

ODV numbers 3-13
 OFL condition
 See overflow condition
 ON conditions 3-17
 See also conditions
 examples 3-18
 ON statement 10-2
 conditions in 10-1
 example 10-5
 precedence over implicit action 10-2
 scope 10-2
 second for same condition 10-4
 on-unit
 ending 10-3
 null 10-3
 running 10-3
 scope of names 10-3
 unlabeled begin-block 10-3
 unlabeled simple statement 10-3
 ONCODE built-in function 15-14
 ONCODE values
 See condition codes

ONFILE built-in function 15-15
 ONKEY built-in function 15-15
 OPEN statement 7-11, 11-4–11-8
 processing 11-4
 opening a SYSPRINT file 7-12
 opening and closing files 11-4, 11-8
 CLOSE statement 11-8
 implicit opening 11-6
 OPEN statement 11-4
 opening stream files 7-12
 operating system 1-1
 operational expressions 9-4, 9-16
 operations 9-4, 9-10
 arithmetic 9-5
 bit 9-9
 combination of 9-14
 comparison
 problem data 9-11
 program control data 9-11
 concatenation 9-14
 conversion of 5-28
 infix 9-4
 prefix 9-4
 priority of 9-15
 operators 4-4, 9-5
 arithmetic 4-4
 bit 4-4, 9-9
 comparison 4-4, 9-10
 concatenation 9-14
 infix 9-5, 9-9
 prefix 9-5, 9-9
 priority of 9-15
 string 4-4
 options
 ASSEMBLER 12-36
 BY 13-7
 FILE in CLOSE statement 11-8
 FILE in DELETE, READ, REWRITE, or
 WRITE statement 11-18
 FILE in GET or PUT statement 11-26
 FILE in OPEN statement 11-4
 FROM 11-19
 INPUT 11-4
 INTO 11-18
 KEY 11-20
 KEYFROM 11-21
 KEYTO 11-21
 LINE 11-27

options (*continued*)

LINESIZE 11-5
MAIN 14-3
OPTIONS 7-13, 14-3
OUTPUT 11-4
PAGE 11-26
PAGESIZE 11-6
RECURSIVE 14-3
REENTRANT 14-3
REPEAT 13-7
RETURNS 14-3
SET 11-19
SKIP 11-26
TITLE 11-4
TO 13-7
UNTIL 13-6
UPDATE 11-4
WHILE 13-6
OPTIONS attribute 12-36
options of record data transmission
statements 11-14, 11-15, 11-16—11-22
FILE 11-18
FROM 11-19
INTO 11-18
KEY 11-20
KEYFROM 11-21
KEYTO 11-21
OPTIONS option 7-13
with INDICATORS parameter 7-19
with RECORD parameter 7-16
SET 11-19
OPTIONS option 7-13, 11-12, 14-3
INDICATORS parameter 7-19
KEYSEARCH parameter 7-17
MODIFIED parameter 7-20
NBRKEYFLDS parameter 7-18
POSITION parameter 6-6, 7-17
RECORD parameter 7-15
organization of a data set 11-9
OTHER (OTHERWISE) statement 13-18
OTHERWISE statement (abbr: OTHER) 13-18
output 11-1
OUTPUT attribute 11-10, 11-23, 12-7
OUTPUT element type in %INCLUDE
directive 8-74
OUTPUT option in OPEN statement 11-4

overflow condition D-4, D-12—D-13
override of member definition for a file 6-2
override of PL/I file declarations 6-2
ownership, file 6-3

P

page eject on source program listing
See %PAGE statement
PAGE format item 11-39
PAGE option 11-26
PAGESIZE option in OPEN statement 11-6
pairing
BEGIN and END statement 13-10
DO and END statement 13-10
PROCEDURE and END statement 13-10
parameter 14-2, 12-33, 14-3
and argument, association of 14-9
attribute specification 14-3
attributes 14-3
declaration 14-2
default scope 14-3
descriptor
list 12-33
nonstructure 12-33
structure 12-33
lengths and bounds 14-4
maximum in a procedure 14-2
parameter descriptor list 12-33
parameters 14-1
parentheses () 4-5
percent statements 4-5
performance considerations
commitment control 8-61
period (.) 4-5
physical files 6-4
DDS example 8-3
PIC attribute
See picture
picture
character
- 12-25
after value truncation to zero 12-25
and zero suppression 12-23
B 12-23
conditional 12-23
CR 12-26
credit and debit 12-26
DB 12-26

- picture (*continued*)
 - character (*continued*)
 - decimal point 12-22
 - digit 12-22
 - drifting 12-24
 - drifting currency 12-24
 - drifting sign 12-24
 - insertion 12-23
 - S 12-25
 - sign and currency characters 12-24
 - static 12-24
 - V 12-22, 12-23
 - Z 12-23
 - zero suppression 12-23
 - 9 12-22
 - . 12-23
 - + 12-25
 - \$ 12-25
 - * 12-23
 - / 12-23
 - , 12-23
 - conversion
 - to bit 5-33
 - to character 5-33
 - to decimal fixed-point 12-21
 - specification 12-19, 12-21
- picture data 12-19–12-22
 - assignment 12-22
 - base 12-21
 - character value 12-19
 - conversion to decimal fixed-point 12-21
 - numeric value 12-19
 - precision 12-21
 - scale 12-21
- picture specification 12-19, 12-21, 12-30
 - derived precision 12-21
 - maximum length 12-20
 - range of numeric values 12-21
 - representation as a character value 12-21
- PLICOMMIT built-in subroutine 15-4
 - use of 8-59
 - with record locking 6-8
- PLIDUMP built-in subroutine 3-14, 15-4
 - example 3-15
- PLIIOFDB built-in subroutine 3-17, 15-5
- PLIOPNFDB built-in subroutine 3-17, 15-5
- PLIRCVMSG built-in subroutine 15-5
- PLIRETC built-in subroutine 15-4
- PLIRETV built-in function 15-19
- PLIROLLBACK built-in subroutine 15-5
 - use of 8-60
 - with record locking 6-8
- PLISHUTDN built-in function 15-20
 - PLISHUTDN 15-20
- PL/I keywords 4-3
 - See also under individual keywords
- PL/I source program
 - description 2-1
 - entering SQL statements 2-5
 - PL/I syntax checker 2-2
 - using SEU 2-2
- PL/I syntax checker 2-2
- point of calling a block 4-10
- POINTER attribute 12-30
- pointer built-in functions 12-31
 - ADDR 12-31
 - NULL 12-31
- pointer data and attribute 12-30
- pointer (->) 4-5, 12-30
 - expression 5-20
 - qualification 5-20
 - qualified reference 5-20
 - qualifier 5-20, 9-1
 - value
 - creation 12-30
 - variable 12-42
- pointers 3-12
- POSITION parameter of the OPTIONS
 - option 7-17
 - values
 - FIRST 7-18
 - LAST 7-18
 - NEXT 7-18
 - NXTEQL 7-18
 - NXTUNQ 7-18
 - PREVIOUS 7-18
 - PRVEQL 7-18
 - PRVUNQ 7-18
 - with READ 6-6, 7-17
- precision 12-10
 - attribute 12-10
 - calculation 5-29
 - coded arithmetic 12-10
 - conversion 5-30

- precision (*continued*)
 - default 12-11
- precision attribute 12-10, 12-11
- prefix operation 9-4
- prefix operator 9-1
- PREVIOUS value 7-18
 - See also POSITION parameter of the OPTIONS option
- PRINT attribute 12-7, 11-44, 12-7
- printer file examples
 - printer file DDS 8-50
 - using a printer file 8-50
- printing messages 3-3
- priority of operators 9-15, 9-16
 - changed by parentheses 9-16
- problem data 9-4, 9-11, 12-9
 - arithmetic 9-11
 - bit 12-16
 - character 12-18
 - conversion 5-27
 - operations
 - arithmetic 9-5
 - bit 9-9
 - comparison 9-11
 - concatenation 9-14
- problem data conversion
 - See data conversion
- PROC statement
 - See PROCEDURE statement
- procedure 4-9, 4-10
 - activation 4-11
 - ending 4-11
 - external 4-6
 - internal 4-6
- PROCEDURE statement 14-2
 - pairing with END statement 13-10
- procedures 4-9—4-11
 - function 14-1
 - initial
 - See initial procedure
 - subroutine 14-2
- Process multiple compilation (see %PROCESS directive) 2-16
- processing an on-unit 10-3, 10-4
- processing mode
 - locate 11-11
 - move 11-11
- processing multiple compilations
 - See %PROCESS directive
- program 4-6
 - activation 4-6
 - debugging 3-1
 - elements 4-1
 - ending 4-6, 13-11
 - organization 4-12
 - structure 4-1
- program control data 9-11, 12-2
 - comparison operations 9-10
 - conversion 5-27
 - label 12-31
 - pointer 12-30
- program elements 4-1
- program ending 13-11
- program ending, abnormal 2-22
- program examples
 - CL program for breakpoints 3-5
 - CL program for trace 3-7
 - customer inquiry program 8-23
 - reading from an arrival sequence file by RRN 8-20
 - updating a file with a keyed access path 8-10
 - updating a file with arrival sequence access path 8-7
 - updating an arrival sequence file by RRN 8-17
 - using a printer file 8-50
 - using commitment control 8-65
 - using externally defined indicators 8-43
 - using PLICOMMIT and PLIROLLBACK 8-68
 - using PLIDUMP 3-15
 - using program-defined indicators 8-47
 - using stream I/O 8-56
 - using subfiles 8-29
 - using %INCLUDE 8-79
 - writing to a file with keyed sequence access path 8-5
 - writing to an arrival sequence file by RRN 8-14
- program object
 - abnormal ending 2-22
 - ending 2-22
 - interrupting 2-22
 - running 2-22

program-described data base files 8-2
 program-described device files 8-50
 program-described record formats 6-12
 use of %INCLUDE directive 8-76
 programs 4-6
 PRV value
 See PREVIOUS value
 pseudovariables 15-5
 SUBSTR 15-23
 UNSPEC 15-25
 PTR attribute
 See POINTER attribute
 PUT statement 11-25
 PUT statement control options 11-26
 FILE 11-26
 LINE 11-27
 PAGE 11-26
 SKIP 11-26

Q

qualified reference 5-5
 quantitative restrictions
 See default, maximum, and minimum
 quotation mark
 See apostrophe (')

R

range
 binary floating-point values 12-14
 bit values 12-16
 character values 12-16
 decimal floating-point values 12-13
 picture specification numeric values 12-21
 READ option 11-12
 READ statement 11-11, 11-12
 OPTIONS option
 with INDICATORS parameter 7-19
 with KEYSEARCH parameter 7-17
 with MODIFIED parameter 7-20
 with NBRKEYFLDS parameter 7-18
 with POSITION parameter 6-6, 7-17
 with RECORD parameter 7-15
 READ 7-13, 11-11
 recognition of names 4-12
 RECORD and STREAM attributes 11-3, 12-7
 default 12-7

RECORD attribute 11-3, 12-7
 record condition D-4, D-13
 record data transmission 11-1, 11-3, 11-8—11-22
 OPTIONS option 7-13
 record blocking 11-9
 statements
 DELETE 7-13, 11-16
 processing 11-10
 REWRITE 7-13, 11-15
 WRITE 7-13, 11-14
 RECORD element type in %INCLUDE
 directive 8-74
 record formats 6-5, 6-11
 externally described 6-11
 defining with DDS 6-11
 level checking 6-12
 program-described 6-12
 record locking 6-7, 8-62
 RECORD parameter of the OPTIONS
 option 7-15
 with DELETE 7-17
 with READ 7-15
 with REWRITE 7-16
 with WRITE 7-16
 RECURSIVE option 14-3, 14-11
 recursive procedures 14-11
 redeclaring a built-in name 12-36
 redirection of files 6-3
 REENTRANT option 14-3
 reference 9-1
 array 9-1
 in DO statements 13-6
 qualified 5-5
 scalar 9-1
 structure 9-1
 subscripted qualified 5-6
 references and expressions 9-1, 9-16
 relationship of pointers and based variables 5-22
 relative record number
 program examples 8-14—8-22
 reopening a closed file 11-8
 REPEAT option 13-7
 repeating a string
 See COPY built-in function
 resource independence 11-2
 results of arithmetic operations 9-5, 9-9

- return code D-1
- RETURN statement 14-4
 - for functions 14-4
 - for subroutines 14-4
- RETURNS attribute 12-35—12-36
- RETURNS option 14-3
 - conversion of value 14-3
- REWRITE statement 11-15, 11-16
 - OPTIONS option 7-13
 - with INDICATORS parameter 7-19
 - with RECORD parameter 7-16
- ROUND built-in function 15-20
- RRN
 - See relative record number
- running programs
 - CL command CALL 2-22
 - communication with non-PL/I programs 2-23
 - ending processing 2-22
 - interrupting processing 2-22
 - PL/I statement CALL 2-22

S

- S picture character 12-25
- SAMEKEY built-in function 15-21
- scalar 9-4
 - assignment 13-1
 - data item 5-1
 - expression 9-1
 - reference 9-1
 - value 9-4
 - variable 5-1
- scalar data mapping 5-10, 5-11
- scale 12-10
 - attributes
 - default 12-10
 - FIXED 12-10
 - FLOAT 12-10
 - coded arithmetic data 12-10
- scale factor, precision attribute 12-11
- scope
 - attributes
 - defaults 12-40, 12-41
 - external 4-16, 7-11, 12-40
 - internal 4-16, 7-11, 12-40
 - of automatic variable 5-16, 12-41
 - of based variable 5-16, 12-41
 - of condition handling built-in function
 - values 10-4

- scope (*continued*)
 - of contextual declaration 4-13
 - of established action 10-4
 - of explicit declaration 4-13
 - of external procedure name 12-41
 - of file constant 12-7
 - of internal procedure name 12-41
 - of names 3-11
 - of open files 7-11
 - of static variable 5-16, 12-41
 - of values of condition handling built-in
 - functions 10-4
- scoping of open files 7-11
- security 6-3
 - file authority 6-3
 - file ownership 6-3
- SELECT statement 13-18
- select-groups 13-18
- semicolon (;) 4-5
- SEQL attribute
 - See SEQUENTIAL attribute
- SEQUENTIAL and DIRECT attributes 11-9, 11-10, 12-7
 - default 11-10
- SEQUENTIAL attribute 11-9, 12-7
 - direct access 11-10
 - sequential access 11-10
- SET option 11-12, 11-19—11-20
 - use with BUFSIZE option of the ENVIRONMENT attribute 7-6
- SEU
 - See source entry utility
- sign and currency characters 12-24, 12-25
 - picture character 12-24
 - drifting string
 - and insertion characters 12-25
 - zero suppression 12-25
 - S picture character 12-24
 - + picture character 12-24
 - \$ picture character 12-24
- SIGN built-in function 15-21
- SIGNAL statement 10-4
 - conditions in 10-1
 - example 10-5
- simple do-group 13-6
- simple statements 4-2, 4-2

- simple string constant 12-43
- SIN built-in function 15-22
- SIND built-in function 15-22
- SINH built-in function 15-22
- SKIP format item 11-39, 11-40
- skip lines on source program listing
 - See %SKIP statement
- SKIP option 11-26
- skip to new page 11-26
- slash (/) B-14
- source entry utility 2-1
 - entering source programs 2-1
 - PL/I syntax checker 2-2
- source program
 - description 2-1
 - entering SQL statements 2-5
 - PL/I syntax checker 2-2
 - using SEU 2-2
- spacing format item 11-40
- special characters B-14
- spooling 6-2
 - input 6-2
 - output 6-2
- SQL statements in a PL/I program 2-5
- SQRT built-in function 15-22
- statement label
 - See label prefix
- statements 4-1, 11-10, 14-7
 - ALLOCATE 5-22
 - assignment 4-2, 13-1
 - BEGIN 4-11
 - CALL 14-7
 - CLOSE 11-8
 - compound 4-1, 13-13
 - IF 13-13
 - data transmission 11-2
 - DECLARE 7-1, 12-1
 - definition 4-1
 - DELETE 11-16
 - DO 13-5
 - elements of 4-2
 - END 13-10
 - FREE 5-23
 - GET 11-24
 - GO TO 13-11
 - IF 13-13
 - input and output 11-1
 - ITERATE 13-14
- statements (*continued*)
 - labels 4-1
 - LEAVE 13-15
 - nesting 13-14
 - null 4-2, 13-17
 - ON 10-2
 - OPEN 11-4
 - OTHERWISE 13-18
 - PROCEDURE 14-2
 - PUT 11-25
 - READ 11-11
 - RETURN 14-4
 - REWRITE 11-15
 - SELECT 13-18
 - SIGNAL 10-4
 - simple 4-2
 - STOP 13-20
 - syntax of 4-1
 - types of 4-1
 - WHEN 13-18
 - WRITE 11-14
 - %INCLUDE 2-16
 - %PAGE 2-18
 - %SKIP 2-19
- static 5-16
 - storage
 - allocation 5-16
 - initialization 5-17
 - variables
 - STATIC attribute 5-16
 - static characters 12-24
 - static storage and attribute 5-16, 5-17
 - STG (STORAGE) built-in function 15-22
 - storage 5-15, 5-16, 5-24
 - allocation
 - dynamic 5-16
 - for automatic variables 5-18
 - for based variables 5-19
 - for static variables 5-16
 - static 5-16
 - class
 - automatic 5-16
 - based 5-16
 - default 5-16, 12-41
 - static 5-16
 - control
 - automatic storage 5-18
 - based storage 5-19
 - built-in functions 15-4

- storage (*continued*)
 - control (*continued*)
 - static storage 5-16
 - requirements
 - binary fixed-point value 12-13
 - character value 12-18
 - decimal floating-point value 12-13
 - label value 12-32
- STORAGE built-in function (abbr: STG) 15-22
- storage condition D-5, D-13
- STREAM attribute 11-3, 12-7
- stream data transmission 11-1, 11-3, 11-22—11-44
 - apostrophes, treatment of 11-29
 - blanks
 - insertion of 11-30
 - treatment of 11-29
 - opening stream files 7-12
 - PRINT attribute 12-8
 - program example 8-56
 - statements 11-23
 - GET 11-23, 11-24
 - PUT 11-23, 11-25
 - SYSIN file 11-44
 - SYSPRINT file 7-12, 11-44
 - truncation 11-30
- stream files 7-12
 - opening 7-12
 - program example 8-56
 - use of 8-55
- string
 - data assignment 5-25
- string data
 - padding 5-25
 - truncation 5-25
- string handling built-in functions 15-2
 - BIT 15-8
 - CHARACTER 15-8
 - COPY 15-8
 - INDEX 15-12
 - LENGTH 15-12
 - SUBSTR 15-23
 - TRANSLATE 15-24
 - UNSPEC 15-25
 - VERIFY 15-26
- string length
 - calculation 5-29
 - for automatic variables 5-19
- string operator 4-4
- string variable length 12-16
- stringsize condition D-5, D-13
- structure
 - arrays of 5-5
 - assignment 13-1
 - component, immediate 5-3
 - declaration 5-4, 12-39
 - field 5-3
 - level number 5-4, 12-39
 - logical level 5-3
 - major 5-3
 - mapping 5-12, 5-15
 - combining two units 5-13
 - example 5-13—5-15
 - maximum level number in DECLARE statement 5-5, 12-39
 - minor 5-3
 - parameter descriptor 12-33
 - qualification 5-5
 - reference 9-1
 - variable 5-1
- structure level
 - logical 5-4
 - number 5-4
- structures 5-3, 5-5
- subfiles of display files 6-9
 - example 8-29
- subroutine call 14-7
- subroutines 14-2
 - arguments 14-1
 - calling 14-1
 - data processed 14-1
 - parameters 14-1
 - return from 14-1
 - RETURN statement in 14-4
- subroutines and functions 14-1, 15-5
- subscript list 9-1
- subscripted qualified reference 5-6
- subscripts 5-2, 5-3
- SUBSTR built-in function 15-23
- SUBSTR pseudovisible 15-23
- syntax checker, PL/I 2-2
- syntax diagrams
 - how to read vi
- SYSIN file 11-44

SYSPRINT file 7-12, 11-44
SYSTEM option in ON statement 10-3
system override of member definition for a file 6-2
system override of PL/I file declarations 6-2
system-dependent features of PL/I 7-1, 7-20

T

TAN built-in function 15-24
TAND built-in function 15-24
TANH built-in function 15-24
target 5-29, 5-31, 5-33
 data attributes 5-27
 length calculation 5-29
 precision calculation 5-29
target variable 13-1
test libraries 3-3
THEN unit 13-13
TIME built-in function 15-24
TO option 13-7
traces 3-7
 example 3-7
TRANSLATE built-in function 15-24
TRANSMIT condition D-5, D-13–D-14
 raised by READ 8-64
 raised by READ, WRITE or REWRITE on
 locked record 6-8

U

UNDEFINEDFILE condition D-5, D-14–D-16
 raised by COMMITTABLE option 7-7
 raised by level checking 6-12
 raised by OPEN 7-12
 raised by unsuccessful OPEN or CLOSE 7-11
UNDEFINEDFILE 8-59
unspecifiable conditions 10-2
 conversion 10-2, D-2
 fixedoverflow 10-2, D-3, D-10
 overflow 10-2, D-4, D-12
 record 10-2, D-4, D-13
 storage D-5, D-13
 stringsize 10-2, D-5, D-13
 underflow 10-2, D-6, D-16
 zerodivide 10-2, D-6, D-16
UNTIL option 13-6
UPDATE attribute 11-10, 11-23, 12-7

using messages 3-1
using, displaying, and printing messages 3-1

V

V picture character 12-22
variables 5-1, 12-38
 automatic 5-18
 based 5-19
 binary fixed-point 12-12
 binary floating-point 12-14
 decimal fixed-point 12-12
 decimal floating-point 12-13
 entry 12-32
 label 12-31
 pointer 12-30
 scalar 5-1
 static 5-16

W

WHEN statement 13-18
WHILE option 13-6

Y

– arithmetic operator 9-5, 4-4
– > (pointer) 4-5

Z

Z picture character 12-23

Numerics

9 picture character 12-22

Special Characters

. (period) 4-5
. insertion character 12-23
< comparison operator 9-10, 4-4
< = comparison operator 9-10, 4-4
+ arithmetic operator 9-5, 4-4
+ picture character 12-25
| bit operator 9-9
|| concatenation operator 9-14
|| string operator 4-4
& bit operator 4-4
\$ currency symbol B-14

\$ picture character 12-24, 12-25
* arithmetic operator 9-5, 4-4
* in parameter descriptor list 12-34
* picture character 12-23
*PROCESS directive 2-16
** arithmetic operator 9-5, 4-4
*/ (end comment) 4-5
¬ bit operator 4-4
/ arithmetic operator 9-5, 4-4
/ insertion character 12-23
/* (begin comment) 4-5
, insertion character 12-23
, (comma) 4-5
%INCLUDE directive 2-16, 2-18
 program examples 8-79
 using for external file descriptions 8-73—8-82
%PAGE statement 2-18
%PROCESS directive 2-16, 2-18
%SKIP statement 2-19, 2-22
> comparison operator 9-10, 4-4
> = comparison operator 9-10, 4-4
: (colon) 4-5
number sign B-14
@ commercial "at" sign B-14
= assignment symbol 4-5
= comparison operator 9-10, 4-4

READER'S COMMENT FORM

Please use this form only to identify publication errors or to request changes in publications. Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your nearest IBM branch office. You may use this form to communicate your comments about this publication, its organization, or subject matter **with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.**

- If your comment does not need a reply (for example, pointing out a typing error), check this box and do not include your name and address below. If your comment is applicable, we will include it in the next revision of the manual.
- If you would like a reply, check this box. Be sure to print your name and address below.

Page number(s):

Comment(s):

Please contact your nearest IBM branch office to request additional publications.

Name _____

Company or
Organization _____

Address _____

SC09-1156-00

Reader's Comment Form

Cut or Fold Along Line

Fold and tape

Please Do Not Staple

Fold and tape



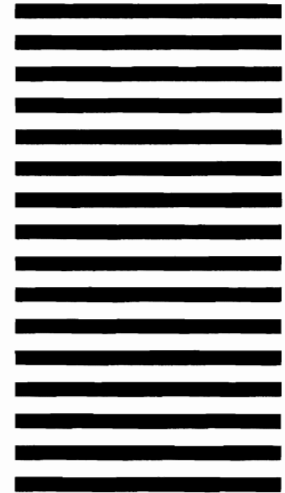
NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS

PERMIT NO. 40

ARMONK, N.Y.



POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Development Laboratory
Information Development, Department 245
Rochester, Minnesota 55901

Fold and tape

Please Do Not Staple

Fold and tape



International Business Machines Corporation

READER'S COMMENT FORM

Please use this form only to identify publication errors or to request changes in publications. Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your nearest IBM branch office. You may use this form to communicate your comments about this publication, its organization, or subject matter **with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.**

- If your comment does not need a reply (for example, pointing out a typing error), check this box and do not include your name and address below. If your comment is applicable, we will include it in the next revision of the manual.
- If you would like a reply, check this box. Be sure to print your name and address below.

Page number(s):

Comment(s):

Please contact your nearest IBM branch office to request additional publications.

Name _____

Company or
Organization _____

Address _____

SC09-1156-00

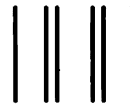
Reader's Comment Form

Cut or Fold Along Line

Fold and tape

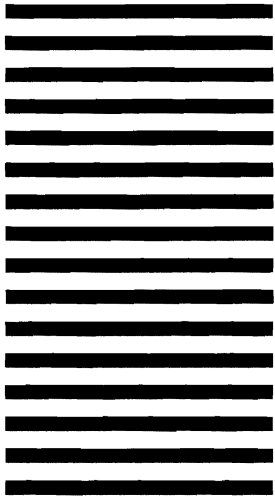
Please Do Not Staple

Fold and tape



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.



POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Development Laboratory
Information Development, Department 245
Rochester, Minnesota 55901

Fold and tape

Please Do Not Staple

Fold and tape



International Business Machines Corporation

READER'S COMMENT FORM

Please use this form only to identify publication errors or to request changes in publications. Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your nearest IBM branch office. You may use this form to communicate your comments about this publication, its organization, or subject matter **with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.**

- If your comment does not need a reply (for example, pointing out a typing error), check this box and do not include your name and address below. If your comment is applicable, we will include it in the next revision of the manual.
- If you would like a reply, check this box. Be sure to print your name and address below.

Page number(s):

Comment(s):

Please contact your nearest IBM branch office to request additional publications.

Name _____

Company or
Organization _____

Address _____

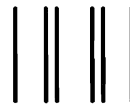
Reader's Comment Form

Cut or Fold Along Line

Fold and tape

Please Do Not Staple

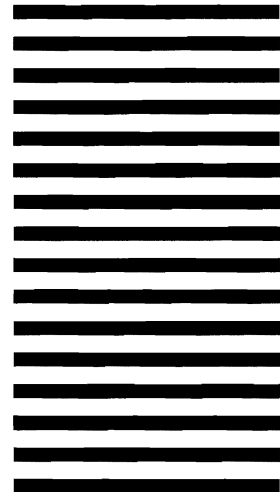
Fold and tape



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.



POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Development Laboratory
Information Development, Department 245
Rochester, Minnesota 55901

Fold and tape

Please Do Not Staple

Fold and tape



International Business Machines Corporation

READER'S COMMENT FORM

Please use this form only to identify publication errors or to request changes in publications. Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your nearest IBM branch office. You may use this form to communicate your comments about this publication, its organization, or subject matter with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

- If your comment does not need a reply (for example, pointing out a typing error), check this box and do not include your name and address below. If your comment is applicable, we will include it in the next revision of the manual.
- If you would like a reply, check this box. Be sure to print your name and address below.

Page number(s):

Comment(s):

Please contact your nearest IBM branch office to request additional publications.

Name _____

Company or
Organization _____

Address _____

SC09-1156-00

Reader's Comment Form

Cut or Fold Along Line

Fold and tape

Please Do Not Staple

Fold and tape



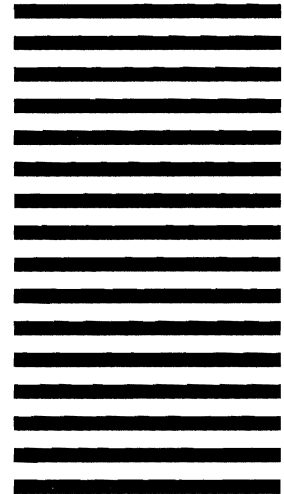
NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS

PERMIT NO. 40

ARMONK, N.Y.



POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Development Laboratory
Information Development, Department 245
Rochester, Minnesota 55901

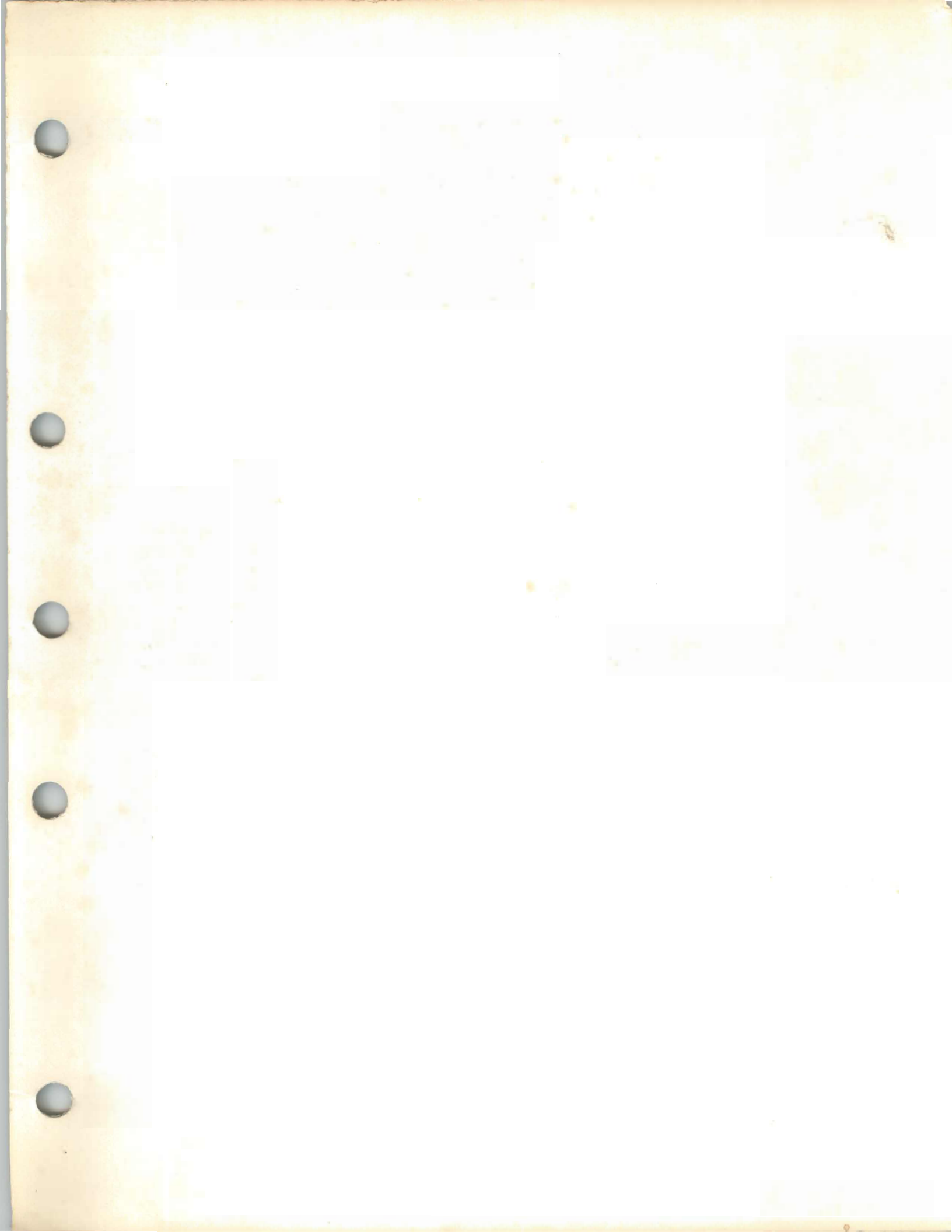
Fold and tape

Please Do Not Staple

Fold and tape



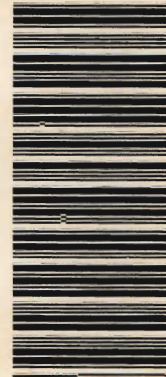
International Business Machines Corporation





Program Number
5728-PL1

21F2753



SC09-1156-00

