

MULTICS PL/I REFERENCE MANUAL

SUBJECT

Reference Manual for the Multics PL/I Language

SPECIAL INSTRUCTIONS

This manual furnishes expanded detail and examples of Multics PL/I language usage as defined in the Multics *PL/I Language Manual*, Order No. AG94.

The reader should also refer to the four manuals that compose the *Multics Programmer's Manual* (MPM) listed below.

<i>Reference Guide</i>	Order No. AG91
<i>Commands and Active Functions</i>	Order No. AG92
<i>Subroutines</i>	Order No. AG93
<i>Subsystem Writers' Guide</i>	Order No. AK92

SOFTWARE SUPPORTED

Multics Software Release 4.0

ORDER NUMBER

AM83, Rev. 0

June 1976

Honeywell

PREFACE

This manual is a combined tutorial and reference manual for Multics PL/I. The manual is intended for a programmer who is experienced in the use of high level languages, such as FORTRAN, COBOL, or ALGOL.

The sections of this manual are organized to reflect the main features of the language. After the introductory section, three aspects of data values are considered: values as abstractions, values in storage, and the conversion of values from one storage type to another. The sections that cover this material are:

- II. Values
- III. Value Storage
- IV. Value Conversion

Next, the overall syntax of a program is considered, ranging from small constructs (such as the identifier) to intermediate constructs (the statements) to large constructs (the blocks). Against this background, the declaration of identifiers and the management of storage is described. The sections are:

- V. Program Structure
- VI. Declaration
- VII. Storage Management

Next, the features that are used to compute and store values are considered. The sections are:

- VIII. Expressions
- IX. Operations
- X. Value Assignment

Next, the features that are used to determine the sequence in which program statements are executed are described. The sections are:

- XI. Program Flow
- XII. Procedure Invocation
- XIII. Condition Handling

Next, the statements for input/output are described. The sections are:

- XIV. Stream Input/Output
- XV. Record Input/Output

Finally, the aspects of Multics that are of special interest to the PL/I programmer are described. The section is:

- XVI. PL/I in the Multics System

The manual has an appendix that gives the syntax of all of the statements of PL/I, except the 'default' statement.

The information and specifications in this document are subject to change without notice. This document contains information about Honeywell products or services that may not be available outside the United States. Consult your Honeywell Marketing Representative.

Related Manuals

Two related Honeywell publications are the PL/I Language Specification and the Multics Programmers' Manual. The PL/I Language Specification is the ultimate authority on Multics PL/I. For the programmer who is beginning a study of PL/I, the manual may be difficult to use and understand because it is designed to be a definition, not an explanation, of PL/I. To the programmer who has learned Multics PL/I, however, the Language Specification is the source of all necessary information about Multics PL/I.

The PL/I Language Specification describes the meaning of any given program, but it does not explain how to write a program. Therefore, it is essential to approach the manual with a well-formulated question, a question that is usually in the form "what does the following PL/I construct mean: ...?" If it is impossible to come up with such a question, then the appropriate section of this manual should be used to get the rules, examples, or guidelines necessary to formulate a question.

The Multics Programmers' Manual (MPM) is actually composed of six manuals: the Reference Guide, Commands and Active Functions, Subroutines, the Subsystem Writers' Guide, Peripheral Input/Output, and Communications Input/Output. For convenience, these references will be as follows:

<u>Document</u>	<u>Referred To In Text As</u>
<u>Reference Guide</u> (Order No. AG91)	MPM Reference Guide
<u>Commands and Active Functions</u> (Order No. AG92)	MPM Commands
<u>Subroutines</u> (Order No. AG93)	MPM Subroutines
<u>Subsystem Writers' Guide</u> (Order No. AK92)	MPM Subsystem Writers' Guide
<u>Peripheral Input/Output</u> (Order No. AX49)	MPM Peripheral I/O
<u>Communications Input/Output</u> (Order No. CC92)	MPM Communications I/O

The MPM Reference Guide contains general information about the Multics command and programming environments. It also defines items used throughout the rest of the MPM. In addition, it describes such subjects as the command language, the storage system, and the input/output system.

The MPM Commands is organized into four sections. Section I contains a list of the Multics command repertoire, arranged functionally. Section II describes the active functions. Section III contains descriptions of standard Multics commands, including the calling sequence and usage of each command. Section IV describes the requests used to gain access to the system.

The MPM Subroutines is organized into three sections. Section I contains a list of the subroutine repertoire, arranged functionally. Section II contains descriptions of the standard Multics subroutines, including the declare statement, the calling sequence, and usage of each. Section III contains the descriptions of some of the I/O modules. Other I/O modules will be described in the MPM Peripheral I/O and the MPM Communications I/O.

The MPM Subsystem Writers' Guide is a reference of interest to compiler writers and writers of sophisticated subsystems. It documents user-accessible modules that allow the user to bypass standard Multics facilities. The interfaces thus documented are a level deeper into the system than those required by the majority of users.

The MPM Peripheral I/O manual contains descriptions of commands and subroutines used to perform peripheral I/O. Included in this manual are commands and subroutines that manipulate tapes and disks as I/O devices.

The MPM Communications I/O manual contains descriptions of commands and subroutines used to perform communications I/O including information on terminal types. Special purpose communications I/O, such as binary synchronous communication, is also included.

The subroutines described in the MPM Subroutines and the MPM Subsystem Writers' Guide can be called from PL/I programs to perform system-provided application and supervisory functions. The I/O modules described in the MPM Subroutines and the MPM I/O manuals can be invoked by calling the `iox_` subroutine to interface directly with the Multics input/output system when performing I/O operations. These I/O module descriptions may also be useful when attaching PL/I files to a disk or tape dataset or other special devices.

CONTENTS

		Page
Section I	Introduction	1-1
	Language Features of PL/I	1-2
	Data Description	1-2
	Program Structure	1-3
	Computation	1-4
	Flow of Control	1-4
	Input/Output	1-5
	Applications of PL/I	1-5
	Scientific Programming	1-5
	Business Programming	1-7
	System Programming	1-8
	Data	1-8
	Program Structure	1-8
	Efficiency	1-9
	Program Validity	1-10
	Examples of Invalid Programs	1-10
	Interpretation of Invalid Programs	1-12
	Suggestions for the Study of PL/I	1-12
	Introductory Texts	1-12
	Contents of this Manual	1-13
Section II	Values	2-1
	Arithmetic Values	2-1
	String Values	2-2
	Address Values	2-3
	Statement Values	2-3
	Locator Values	2-4
	File Values	2-4
	Area Values	2-4
	Aggregate Values	2-4
	Classification of Values	2-5
Section III	Value Storage	3-1
	Storage Units	3-1
	Storage Types	3-2
	Arithmetic Storage	3-4
	Arithmetic Data Types	3-5
	Mode Attribute	3-5
	Scale Attribute	3-5
	Base Attribute	3-6
	Precision Attribute	3-7
	Abbreviations and Defaults	3-8
	Examples of Arithmetic Storage Units	3-10
	Fixed Decimal Storage Units	3-11
	Fixed Binary Storage Units	3-12
	Float Binary Storage Units	3-13
	Float Decimal Storage Units	3-13
	Complex Storage Units	3-14
	Guidelines for Using Arithmetic Data Types	3-14
	Choice of Mode	3-14
	Choice of Scale and Base	3-15
	Choice of Precision	3-15

	Page
Ordinary String Storage	3-16
Ordinary String Data Types	3-16
String-type Attribute	3-16
Variability Attribute	3-17
Abbreviations and Defaults	3-17
Examples of Ordinary String Storage	3-18
Character Storage Units	3-18
Bit Storage Units	3-19
Guidelines for Using Ordinary String Data Types	3-19
Pictured String Storage	3-20
Pictured Data Types	3-20
Picture Attribute	3-21
Classification of Indicators	3-21
Classification of Pictures	3-22
Mode Attribute	3-23
Abbreviations and Defaults	3-23
Interpreting Pictured Storage	3-23
Character-string Interpretation of Pictured Storage	3-24
Related Character Types	3-24
Character-string Assignments	3-25
Character-string Fetches	3-26
Arithmetic Interpretation of Pictured Storage	3-26
Related Arithmetic Data Types	3-26
Arithmetic Assignments	3-27
Arithmetic Fetches	3-28
Fixed-point Pictures	3-29
No-suppression Digit Indicator	3-30
Decimal-point Indicator	3-32
Sign Indicators	3-33
Dollar Indicator	3-34
Zero-suppression Digit Indicators	3-35
Drifting-sign Digit Indicators	3-36
Drifting-dollar Digit Indicator	3-37
Insertion-character Indicators	3-38
Arithmetic Decimal-point Indicator	3-39
Scale-factor Indicator	3-40
Floating-point Pictures	3-41
Floating-point Indicators	3-41
Scale-factor Indicator	3-42
Complex Pictures	3-42
Non-numeric Pictures	3-43
Non-numeric Indicators	3-43
Guidelines for Pictured Storage	3-44
Address Storage	3-44
Address Attributes	3-45
Example of Address Storage	3-46
Area Storage	3-46
'area' Attribute	3-47
Default Rule	3-48
Example of Area Storage	3-48
Aggregate Storage	3-49
Structures	3-49
Level Numbers	3-49
Structure Storage Types	3-50
Examples	3-51
Arrays	3-53
'dimension' Attribute	3-54
Abbreviations and Defaults	3-55

CONTENTS (cont)

	Page
Array Storage Types	3-56
Examples	3-57
Guidelines for Aggregates	3-60
Alignment	3-61
Alignment Attribute	3-62
Abbreviations and Defaults	3-63
Storage Layout Rules for Multics	3-66
Storage Layout for Scalars	3-66
Storage Layout for Structures	3-68
Storage Layout for Arrays	3-71
Section IV	
Value Conversion	4-1
Contexts that Force Conversion	4-1
General Contexts	4-2
Assignment Statements	4-2
Assignment-Like Constructs	4-3
Arguments and Results	4-3
Built-In Functions and Expressions	4-4
Special Contexts	4-4
Integer Targets	4-5
Character-String Targets	4-5
Bit-String Targets	4-6
Locator Targets	4-6
Data Type Conversion	4-7
Arithmetic Targets	4-7
Examples of Arithmetic to Arithmetic Conversion	4-9
Examples of String to Arithmetic Conversion	4-10
Character-String Targets	4-10
Examples of Character-String to Character-String Conversion	4-12
Examples of Bit-String to Character-String Conversion	4-12
Examples of Arithmetic to Character-String Conversion	4-13
Float Decimal Values	4-13
Fixed Decimal Values	4-13
Float Binary Values	4-14
Fixed Binary Values	4-15
Complex Values	4-15
Bit-String Targets	4-16
Examples of Bit-String to Bit-String Conversion	4-16
Examples of Character-String to Bit-String Conversion	4-17
Examples of Arithmetic to Bit-String Conversion	4-17
Fixed Binary Values	4-17
Fixed Decimal Values	4-17
Float Values	4-18
Complex Values	4-18
Locator Targets	4-18
Aggregate Type Conversion	4-18
Example of Aggregate Conversion	4-19
Conditions for Conversions	4-20
'size' and 'fixedoverflow' Condition	4-20
'overflow' and 'underflow' Conditions	4-21
'conversion' Condition	4-21
'stringsize' Condition	4-22
Guidelines for Conversion Conditions	4-22

CONTENTS (cont)

Page

Section V

Program Syntax 5-1
 Characters 5-2
 Lexemes 5-2
 Identifiers 5-3
 Keyword vs. Name 5-3
 Guidelines for Identifiers 5-4
 Literal Constants 5-5
 Punctuators 5-6
 Operators 5-6
 Pictures 5-7
 Isubs 5-7
 '%include' Macro 5-7
 Macro Interpretation 5-8
 Macro Example 5-8
 Guidelines for Macros 5-9
 Separators 5-10
 Separation Rules 5-10
 Classification of Lexemes 5-12
 Statements 5-13
 Statement Prefix 5-13
 Condition Prefixes 5-13
 Label Prefixes 5-13
 Statement Body 5-14
 Classification of Statements 5-16
 Program Structures 5-17
 Groups 5-17
 Blocks 5-18
 Summary of the Program Structures 5-19
 External Procedures and the Program 5-20

Section VI

Declarations 6-1
 Establishment of Declarations 6-1
 Example of the Establishment of Declarations 6-2
 Containment and Immediate Containment 6-3
 'declare' Statement 6-4
 Simple Declarations 6-4
 Structure Declarations 6-5
 Short Forms of Declarations 6-6
 Abbreviations and Defaults 6-6
 Combining Declarations 6-6
 Factoring Declarations 6-6
 Guidelines for 'declare' Statements 6-7
 Label Prefixes 6-8
 Label Constant Names 6-8
 Entry Constant Names 6-8
 Format Constant Names 6-9
 Contextual and Implicit Declarations 6-10
 Example of Contextual and
 Implicit Declarations 6-10
 Guidelines for Contextual and
 Implicit Declarations 6-11
 Special Facilities for Declaration 6-11
 'like' Attribute 6-11
 Form of the 'like' Attribute 6-12
 Interpretation of the 'like' Attribute 6-12
 Examples of the 'like' Attribute 6-13
 Guidelines for the 'like' Attribute 6-13
 'default' Statement 6-13
 Examples of the 'default' Statement 6-14
 Guidelines for the 'default' Statement 6-15
 Resolution of Name References 6-15

CONTENTS (cont)

	Page
Example of the Resolution of Name References.	6-16
Name-Sequence Set for a Declaration	6-16
Name-Sequence for a Name Reference.	6-17
Applicability of Declarations	6-17
Resolution Rules.	6-18
Attributes	6-20
Complete Attribute Sets	6-20
Variable Names	6-21
Constant Names	6-22
Built-in Function Names.	6-23
Condition Names.	6-23
Generic Names.	6-23
Classification of Attributes.	6-24
 Section VII	
Storage Management.	7-1
Preliminary Examples of Storage Management	7-2
Fundamentals of Storage Management	7-3
Storage Regions	7-4
Region Diagrams	7-5
Storage Management Operations	7-6
Storage Management Example.	7-6
Management Class	7-10
Form of the Management Class.	7-10
Abbreviations and Defaults.	7-11
Default Rules.	7-12
Usage Categories	7-12
Variables	7-13
Constants	7-13
Scopes	7-13
Internal Names.	7-13
External Names.	7-14
Guidelines for the Scope.	7-14
Correspondence Between Names and Storage.	7-14
Storage Classes.	7-15
Automatic Variables	7-16
Variable Expressions in Attributes for 'automatic' Variables	7-16
Saving Variable Extents for 'automatic' Variables	7-17
Static Variables.	7-17
Variable Expressions in Attributes for 'static' Variables.	7-18
Controlled Variables.	7-18
'allocate' and 'free' Statements for 'controlled' Variables	7-18
Stacking Controlled Variables	7-18
Variable Expressions in Attributes for 'controlled' Variables.	7-19
Saving Variable Extents for 'controlled' Variables	7-20
Parameter Variables	7-20
Variable Expressions in Attributes for 'parameter' Variables	7-20
Based Variables	7-20
Reference in the 'based' Attribute	7-22
'allocate' and 'free' Statements for 'based' Variables	7-22
Variable Expressions in Attributes	7-23
Saving of Variable Extents	7-23
'refer' Option	7-24
Equivalenced Based Variables	7-25

CONTENTS (cont)

Page

Simple Based Variables 7-26
 String Overlay Based Variables 7-27
 Partial Based Variables 7-28
 Defined Variables 7-29
 Variable Expressions in Attributes for
 'defined' Variables 7-29
 Saving Variable Extents for 'defined'
 Variables 7-30
 Uses of Defined Variables 7-30
 Simple Defining 7-30
 String Overlay Defining 7-31
 Isub Defining 7-31
 Guidelines for the Storage Class 7-32
 Automatic Variables 7-32
 Static Variables 7-33
 Controlled Variables 7-33
 Parameter Variables 7-34
 Based Variables 7-34
 Defined Variables 7-34
 'initial' Attribute 7-35
 Initialization Syntax 7-35
 Use of 'initial' Attribute 7-36
 'options(constant)' Attribute 7-37
 Capacity of Storage 7-38
 Storage Limits 7-38
 Significant Uses of Storage 7-39
 Conditions for Storage Management 7-39
 'storage' Condition 7-40
 'area' Condition 7-40

Section VIII

Expressions 8-1
 General Remarks on Expressions 8-1
 Nested Expressions 8-2
 Parenthesized Expressions 8-3
 Storage Types of Expressions 8-3
 Aggregate Expressions 8-5
 Ordering and Optimizing the Evaluation
 of Expressions 8-5
 Variable References 8-7
 Variable Reference Types 8-7
 Major Name in a Variable Reference 8-7
 Simple Variable References 8-8
 Form of Simple Variable References 8-8
 Interpretation of Simple Variable
 References 8-9
 Examples of Simple Variable References 8-10
 Subscripted Variable References 8-11
 Form of Subscripted Variable References 8-11
 Interpretation of Subscripted Variable
 References 8-11
 Examples of Subscripted Variable
 References 8-13
 Structure-Qualified Variable References 8-15
 Form of Structure-Qualified Variable
 References 8-15
 Interpretation of Structure-Qualified
 Variable References 8-16
 Examples of Structure-Qualified Variable
 References 8-17
 Locator-Qualified Variable References 8-20
 Form of a Locator-Qualified Variable

CONTENTS (cont)

	Page
Reference	8-21
Associated Storage Types for Locator Values.	8-21
Interpretation of Locator-Qualified Variable References	8-22
Examples of Locator-Qualified Variable References.	8-23
Shortened Forms of References	8-25
Subscript-List Deletion.	8-25
Examples of Subscript List Deletion	8-26
Guidelines for Subscript List Deletion.	8-26
Name Deletion.	8-26
Examples of Name Deletion	8-27
Guidelines for Name Deletion.	8-28
Locator Qualifier Deletion	8-28
Guidelines for Locator-Qualifier Deletion	8-28
Cost of Variable References	8-29
Constant Literals.	8-29
Arithmetic Constant Literals.	8-29
Form of Arithmetic Constant Literals	8-30
Interpretation of Arithmetic Constant Literals.	8-31
Examples of Arithmetic Constant Literals	8-31
Guidelines for Arithmetic Constant Literals.	8-32
String Constant Literals.	8-32
Form of String Constant Literals	8-32
Interpretation of String Constant Literals	8-33
Examples of String Constant Literals	8-33
Escape Conventions for Characters.	8-34
Attributes for Constant Literals.	8-34
Constant References.	8-35
Statement Constant References	8-35
Form of Statement Constant References.	8-35
Interpretation of Statement Constant References.	8-36
Examples of Statement Constant References.	8-36
External 'entry' Constant References	8-37
File Constant References.	8-38
Form of File Constant References	8-38
Examples of File Constant References	8-38
Attributes for Constant Names	8-39
Programmed Function References	8-39
Form of Programmed Function References.	8-39
Interpretation of Programmed Function References	8-40
Examples of Programmed Function References.	8-40
Built-in Function References	8-42
Form of Built-in Function References.	8-43
Interpretation of Built-in Function References	8-43
Examples of Built-in Function References.	8-44
Differences Between Built-in and Programmed Function References.	8-45
Operator Expressions	8-46
Form of Operator Expressions.	8-47
Interpretation of Operator Expressions.	8-47
Operator Priority Rules	8-48
Examples of Operator Expressions.	8-49

CONTENTS (cont)

Page

Section IX

Operations.	9-1
General Remarks.	9-2
Argument Evaluation	9-2
Non-Standard Operations	9-3
Conventions for Examples.	9-3
Arithmetic Operations.	9-3
Conventions for Definitions	9-4
Arguments.	9-4
Common Data Attributes	9-5
Elementary Operations	9-5
Prefix Sign Operators.	9-6
Infix Sign Operators	9-6
Multiplication Operator.	9-7
Division Operator.	9-8
Exponentiation Operator.	9-9
'add', 'subtract', 'multiply', and 'divide' Functions.	9-10
Comparison Operations	9-10
Relational Operators for Arithmetic Values.	9-10
'min' and 'max' Functions.	9-11
Truncating Functions.	9-12
'trunc', 'floor', and 'ceil' Functions	9-12
'round' Function	9-13
'mod' Function	9-14
Sign-Manipulation Functions	9-16
'abs' Function	9-16
'sign' Function.	9-17
Complex Arithmetic Functions.	9-17
'real' and 'imag' Functions for Arithmetic	9-18
'complex' Function	9-18
'conjg' Function	9-19
Mathematical Operations.	9-19
Conventions for Definitions	9-19
Arguments.	9-19
Results.	9-20
Evaluation	9-20
Functions Related to Exponentiation	9-21
'exp' Function	9-21
'log' Function	9-22
'log10' and 'log2' Functions	9-22
'sqrt' Function.	9-23
Trigonometric Functions	9-24
'sin', 'cos', and 'tan' Function	9-24
Ordinary 'acos' Function	9-24.1
Ordinary 'asin' Function	9-24.1
'sind', 'cosd', and 'tand' Functions	9-25
Ordinary 'atan' Function	9-25
Ordinary 'atand' Function.	9-26
Cartesian 'atan' Function.	9-26
Cartesian 'atand' Function	9-27
Hyperbolic Functions.	9-28
'sinh', 'cosh', and 'tanh' Functions	9-28
'atanh' Function	9-29
Functions for Statistical Analysis.	9-29
'erf' and 'erfc' Functions	9-30
String Operations.	9-30
Conventions for Definitions	9-30
Arguments.	9-31
Concatenate Operations.	9-32
Concatenate Operator	9-32

CONTENTS (cont)

	Page
'copy' Function	9-32
Substring Operations	9-33
'substr' Function	9-33
'index' Function	9-34
'before' Function	9-34
'after' Function	9-35
'decat' Function	9-36
Relational, Length, and Reverse Operations	9-37
Relational Operators for String Values	9-38
'length' Function	9-40
'maxlength' Function	9-40
'reverse' Function	9-40.1
'ltrim' Function	9-40.1
'rtrim' Function	9-40.1
Bit-String Operations	9-41
Logical Operators	9-41
'bool' Function	9-42
Character-String Operations	9-43
'search' Function	9-43
'verify' Function	9-43
'translate' Function	9-44
Character-Set Operations	9-45
'collate' Function	9-45
'collate9' Function	9-47
'low' Function	9-47
'high' Function	9-47
'high9' Function	9-48
String Functions Defined Elsewhere	9-48
Address and Area Operations	9-48
General Address Functions	9-49
Relational Operations for Address Values	9-49
'addr' Function	9-50
'null' Function	9-51
'nullo' Function	9-51
Implementation-Dependent Address Functions	9-52
'baseptr' Function	9-52
Nonstandard 'ptr' and 'addrel' Functions	9-52.1
'baseno' and 'rel' Functions	9-52.2
'stackframeptr' Function	9-52.2
'stackbaseptr' Function	9-52.2
'codeptr' Function	9-52.2
'environmentptr' Function	9-52.2
Area Function	9-53
'empty' Function	9-53
Array Operations	9-53
Extent Functions	9-53
'lbound' and 'hbound' Functions	9-54
'dimension' Function	9-54
Special Array Functions	9-55
'sum' and 'prod' Function	9-55
'dot' Function	9-56
Conversion Operations	9-56
Fundamental Conversion Function	9-57
'convert' Function	9-57
Conversion to Arithmetic Data Types	9-59
'real' Function for Conversion	9-59
'complex' Function for Conversion	9-60
'fixed' Function	9-60
'float' Function	9-61
'binary' Function	9-62
'decimal' Function	9-63

CONTENTS (cont)

Page

'precision' Function	9-64
Conversion to String Data Types	9-64
'character' Function	9-64
'bit' Function	9-65
Conversion between Locative Data Types.	9-66
Standard 'pointer' Function.	9-66
'offset' Function.	9-66
Special Conversion Functions.	9-66
'string' Function.	9-67
'unspec' Function.	9-68
'valid' Function	9-69
Guidelines for Conversion Functions	9-69
System Variable Operations	9-70
System Counter Functions.	9-71
'lineno' and 'pageno' Functions.	9-71
'clock' and 'vclock' Functions	9-71
'time' and 'date' Functions.	9-71
Storage Management Functions.	9-72
'allocation' Function.	9-72
'size' Function.	9-73
'currentsize' Function	9-73
Resource Reservation Function	9-74
'stac' Function	9-74
'stacq' Function	9-74.1
'on' Condition Functions.	9-75
'onloc' Function	9-76
'oncode' Function.	9-76
'onkey' Function	9-77
'onfield' Function	9-77
'onchar' and 'onsource' Functions.	9-78
'onfile' Function.	9-79

Section X

Value Assignment.	10-1
Examples of Assignment Statements.	10-1
Arithmetic Assignment Statements.	10-1
String Assignment Statements.	10-2
Address Assignment Statements	10-2
Area Assignment Statements.	10-3
Aggregate Assignment Statements	10-3
Form of Assignment Statements.	10-4
Targets	10-4
Interpretation of Assignment Statements.	10-5
Special Restrictions.	10-5
Overlapping String Targets	10-5
Area Assignments	10-7
Order of Interpretation	10-7
Pseudo-Variables	10-9
'real' and 'imag' Pseudo-Variables.	10-9
'substr' as a Pseudo-Variable	10-10
'string' Pseudo-Variable.	10-11
'unspec' Pseudo-Variable.	10-12
'pageno' Pseudo-Variable.	10-12
'onchar' and 'onsource' Pseudo-Variables.	10-13

Section XI

Program Flow.	11-1
Sequential Execution	11-2
'if' Statement	11-3
Test in an 'if' Statement	11-4
Consequences in an 'if' Statement	11-4
Non-iterative 'do' Group as a Consequence	11-6

CONTENTS (cont)

	Page
'if' Statement as a Consequence	11-6
Dangling 'else' Clause	11-7
'do' Group	11-8
Iterative 'do' without Index	11-9
Iterative 'do' with Index	11-10
Single-Value Control	11-11
'repeat' Control	11-12
FORTRAN Control	11-13
Index of a 'do' Group	11-15
Non-iterative 'do'	11-16
'goto' Statement	11-16
'goto' with a Constant Reference	11-16
'goto' with a Non-Constant Reference	11-17
'local' Attribute	11-17
Restriction on the Destination	11-18
Block Execution	11-19
Guidelines for Flow of Control	11-19
Avoidance of Unnecessary 'goto' Statements	11-19
Layout Conventions	11-20
 Section XII	
Procedure Invocation	12-1
Arguments and Parameters	12-2
Argument Classification	12-3
Examples of Argument Classification	12-4
Examples of Connected and Unconnected Arguments	12-5
Argument Interpretation	12-6
Examples of Argument Interpretation	12-6
Effect of an '*' Extent	12-7
Example of an '*' Extent	12-7
Guidelines for Arguments	12-8
'call' Statement	12-9
Agreement of Call with Entry Point	12-9
Execution of 'call' Statements	12-9
Interpret Entry Reference	12-10
Interpret Arguments	12-10
Activate Procedure	12-10
Execute Procedure	12-11
Exit from Procedure	12-11
Examples of 'call' Statements	12-12
Function References	12-14
Agreement of Reference with Entry Point	12-14
Interpretation of Function References	12-15
Exit from Procedure	12-15
Fetch Result	12-16
Example of a Function Reference	12-16
Procedures	12-17
'procedure' Statement	12-17
Prefix	12-18
Parameter List	12-19
'returns' Attribute	12-19
Example of the 'returns' Attribute	12-20
'recursive' Keyword	12-20
'entry' Statement	12-21
Example of a Multiple-Entry Procedure	12-21
'return' Statement	12-22
'end' Statement	12-22
Entry References	12-22
Constant Entry References	12-23
Entry Points in the Same External Procedure	12-23
Entry Points in Another External Procedure	12-23

CONTENTS (cont)

Page

Entry Points Outside the Program	12-24
'options' Attribute	12-24
Variable Entry References	12-25
Example of an Entry Variable Reference	12-25
Function Entry Reference.	12-26
Example of an Entry Function Reference	12-26
Generic Entry Names	12-27
Example of a Generic Entry Name.	12-27
Recursive Procedure Execution.	12-28
Example of Recursion without Arguments	12-29
Example of Recursion with an Argument.	12-31
Example of Chained Recursion	12-32
Recursive Program.	12-33
General Recursion	12-34
Activation Indexes	12-35
Pointers	12-36
Example of Pointers	12-36
Parent Designators	12-37
Activation Variable References.	12-38
Statement Address Constant References	12-39
Setting the Parent Designator.	12-39
'procedure' Block	12-39
'begin' Block	12-40
'format' Statement.	12-40
'on' Unit	12-41
'goto' Statement	12-42

Section XIII

Condition Handling.	13-1
Principal Features of Condition Handling	13-1
Conditions	13-2
Language-Defined Conditions	13-2
Computational Conditions	13-2
Storage Conditions	13-3
Termination Conditions	13-4
Input/Output Conditions.	13-4
Programmer-Defined Conditions	13-5
Condition References	13-6
Language-Defined Condition References	13-6
Programmer-Defined Condition References	13-6
Declaration of Condition Names	13-7
Enabling and Disabling Conditions.	13-7
Condition Prefixes.	13-7
Scope of a Condition Prefix	13-8
Default Enabling and Disabling.	13-9
Example of Enabling and Disabling	13-10
Establishing and Reverting 'on' Units.	13-10
'on' Statement.	13-11
'revert' Statement.	13-12
Occurrence and Signalling of Conditions.	13-12
Occurrence of Conditions.	13-12
'signal' Statement.	13-13
Determining the Established 'on' Unit	13-14
'on' Unit.	13-14
Default 'on' Units.	13-15
'on' Condition Built-in Functions	13-15
Example of Condition Handling.	13-17
Guidelines for Condition Handling.	13-17
Debugging	13-18
Enabling Conditions for Debugging.	13-18
'on' Units for Debugging	13-18
Controlling Exceptional Conditions.	13-18

CONTENTS (cont)

	Page
Controlling File Communication Conditions	13-18
Controlling Error Conditions	13-19
Input Data Validation	13-19
Computational Checks	13-19
Resource Management	13-19
Large Independent Systems	13-19
General Conditions	13-19
'error' Condition	13-20
'finish' Condition	13-20
 Section XIV	
Stream Input/Output	14-1
Stream Data Sets	14-1
Control Characters in PL/I and Multics	14-2
Input Streams	14-3
Output Streams	14-3
Pseudo-Streams	14-4
Multics Files	14-4
Stream Input/Output Files	14-4
File-State Blocks	14-5
File References	14-6
File Attachment	14-7
Attaching a Switch	14-7
Attach Description	14-7
Opening a Switch	14-8
Opening a File	14-8
Stream Input/Output Operations	14-8
Opening and Closing Files	14-9
'open' Statement	14-10
'close' Statement	14-11
Default Files	14-12
Input/Output Statements	14-13
'get' Statement	14-13
'put' Statement	14-14
'format' Statement	14-15
Data-Directed Input/Output	14-16
Examples of Data-Directed Input/Output	14-16
Principles and Exceptions	14-19
Guidelines for Data-Directed Input/Output	14-20
List-Directed Input/Output	14-21
Example of List-Directed Input/Output	14-21
Compound List Items	14-22
Pseudo-Variable List Items	14-24
Principles and Exceptions	14-25
Guidelines for List-Directed Input/Output	14-25
Edit-Directed Input/Output	14-26
Examples of Edit-Directed Input/Output	14-26
Data Format Items	14-28
String Format Items	14-29
String Input	14-30
String Output	14-30
Fixed-Point Format Items	14-30
Fixed-Point Input	14-31
Fixed-Point Output	14-32
Floating-Point Format Items	14-32
Floating-Point Input	14-33
Floating-Point Output	14-33
Complex Format Items	14-34
Complex Input	14-34
Complex Output	14-35
Picture Format Items	14-35
Fixed-Point Pictures	14-36

SECTION I

INTRODUCTION

PL/I is a general-purpose, high-level programming language. It is designed for use across the entire spectrum of computer applications, including scientific, business, and system programming; and it is an alternative to FORTRAN, COBOL, or assembly language. The complete PL/I language is a large and complicated language that provides an experienced programmer with unusual power and flexibility. On the other hand, subsets of PL/I can be selected for specific application areas, and these subsets are easy to learn and use.

PL/I has a wide range of data types and data structures, and these allow program data to be organized in a clear and convenient way. The program syntax and the control statements of PL/I allow programs to be written in a modular, structured, and easy-to-read style.

Multics PL/I is closely related to American National Standards Programming Language PL/I. When this manual mentions Standard PL/I, the reference is to the language described in ANSI X3.53-1976. A reference to any of the non-Standard functions makes a program dependent on the data representation of Multics PL/I. This manual does not specify all differences between Multics PL/I and Standard PL/I. For a complete description of the differences between Multics PL/I and Standard PL/I, see Appendix A of the PL/I Language Specification.

This manual covers all of Multics PL/I. Each feature of the language is explained by an example, and the rules of the language are given in definitions that are informal but complete.

Most of this section is devoted to a general description of PL/I. The description begins with a listing of the main features of the language and continues with a consideration of the applications of PL/I. After that, the fundamental notions of program validity and correctness are defined. Finally, several publications that are useful in the study of PL/I are cited.

LANGUAGE FEATURES OF PL/I

PL/I brings together in a single language some of the most successful features of earlier programming languages. The design of the language was a major undertaking; it took nearly ten years and involved many separate groups and committees. The final result is a consensus rather than a unified approach to programming. The strength of the language is in the great variety of its features; its weakness is in the relation of these features to one another. A description of the main features of the language follows.

Data Description

In PL/I, a variable is described in terms of the set of values it can accommodate rather than in terms of the hardware storage it occupies. Consider the following declarations:

```
dcl x fixed decimal(7,2);
dcl y character(8);
dcl z pointer;
```

Each of the three variables just declared occupies two words of Multics memory; therefore, from the point of view of the hardware, the variables are very similar. However, from the point of view of PL/I, each variable is entirely different from the others. The storage designated by 'x' accommodates a fixed-point number with seven decimal digits, two of which occur after the decimal point. The storage designated by 'y' accommodates a string of eight ASCII characters. The storage designated by 'z' accommodates the address of a variable.

The handling of data storage in the manner just described is fundamental to the nature of high level languages. It makes programs less hardware dependent. In addition, it allows the compiler to detect some errors in a program (such as an attempt to take the square root of a program address) and to supply conversions where they are required (as in the assignment of a fixed-point value to a floating-point variable). The more information the data descriptions give, the more efficiently a compiler can manipulate the data.

PL/I is unusual because of the large number of different storage types that are available in the language. A variable can be declared in any of the following ways:

- An arithmetic variable can be declared to accommodate a number with fixed-point or floating-point scale and binary or decimal base. The length of the number can range from one digit to many digits, and the binary point or decimal point can be positioned anywhere in the number. The number can be complex for use in scientific programming. Therefore a programmer can choose the representation that suits his needs.
- A string variable can be declared to accommodate a sequence of ASCII characters or a sequence of bits. It can be stored as a compact, fixed-length string or a more flexible varying-length string. In Multics, a string can be very long; in fact, a book of considerable size can be stored in a single character-string variable.
- A pictured variable can be used to accommodate a value that can be interpreted either as a number or a character string, as circumstances require. The use of such variables can greatly simplify the formatting of input/output.

- An address variable or an area variable can be used in performing operations that formerly could be performed only in assembly language.
- An array variable can be declared to accommodate a sequence of smaller variables, all of the same type, that are designated by subscripts.
- A structure variable can be declared to accommodate a sequence of smaller variables that are not necessarily of the same type and that are designated by subnames.

Each different kind of data is called a storage type. The large number of storage types in PL/I is the main cause of the size of the language. For each storage type, both the range of values and the representation of values must be defined. For arithmetic and string values, the representation of each value as a character sequence must be defined, so that the value can be read in, printed out, or used as a constant in a program. Rules for conversion must be given for any pair of storage types between which conversion is reasonable. The operators and builtin functions must be defined to operate on many storage types directly, without a preliminary conversion of operands.

A programmer who is learning PL/I needs a casual acquaintance with the full range of storage types. Once the programmer begins to write a specific program, however, he can concentrate on the storage types required by the program. For example, a program can be written that reads in some numbers, performs some computations, and then prints out numbers; such a program requires only arithmetic storage types. Later, a second version might be written that produces output suitable for immediate publication; this version would require character string variables as well as numbers. Still later, the program might be converted to operate on a permanent data base, and address variables and area variables would be useful. An advantage of PL/I is that a program can become more complicated without outgrowing the language.

Program Structure

A PL/I program is a set of one or more external procedures. Each external procedure is compiled separately; nevertheless, any variable can be shared between all the external procedures of a program by declaring it 'external'. This arrangement makes it practical to develop a large program as a collection of separate modules; then, when development is complete, the modules can be used together.

Each external procedure can contain internal procedures. Internal procedures can be nested, so that a given procedure can be programmed in terms of smaller procedures. Each procedure can have its own variables, so that the data as well as the program can be structured by means of nested procedures. This feature of PL/I makes it suitable for top-down program design.

Storage management is closely related to the procedure structure of a program. In most applications, storage management requires little attention from the programmer. When storage management is not specified for a variable, the variable is automatically allocated and freed as control enters and leaves the procedure in which it is declared. Occasionally it is necessary to declare a variable 'static' so that it will remain throughout the Multics process. Advanced features for storage management are available for use where programmed storage management is necessary.

Computation

The computational power of PL/I is provided by the more than one hundred built-in operations of the language. Each operation is designated either by an operator, such as '+', or a built-in function name, such as 'log'. The operations are:

- The arithmetic operations, which include the basic arithmetic operators, the relational operators, and built-in functions for comparison, truncation, sign-manipulation, and complex arithmetic
- The mathematical operations, which are built-in functions for exponentiation, logarithms, trigonometry, and statistical analysis
- The string operations, which include operations for putting strings together, taking them apart, comparing them, searching them, and ordering them
- The address and area operations, which are used for advanced methods of storage management and list processing
- The array operations, which are especially for the manipulation of array variables
- The conversion operations, which can be used to perform any reasonable conversion between storage types
- The special operations, which are used for details of input/output, interrupt handling, and the determination of the time and date.

The interpretation of an operation depends on the storage types of its operands; for example, the '+' operator is interpreted in one way for fixed-point decimal operands, in a very different way for complex floating-point operands, and in yet another way for array operands. It is a fundamental principle of PL/I that if there is a reasonable interpretation for an operator with given operands, then the operator is defined for those operands. Thus in PL/I, as in mathematics, a single operator can have many meanings.

The computation of a value is specified by an expression. Another fundamental principle of PL/I is that an expression can often be used almost anywhere that makes sense. Thus an expression can be used to specify the number of elements in an array, the increment of a 'do' loop, or an output value.

Flow of Control

Generally, the statements of a program are executed in the order in which they appear; however, this sequence can be modified by flow-of-control statements, by procedure invocation, or by the occurrence of conditions. Each of these methods for modifying the sequence of execution is considered briefly here.

There are three statements that alter the flow of control of a program. The 'if' statement causes conditional execution of a statement or a group of statements; the syntax of the statement allows the logic of a program to be laid out in a clear and readable way. The 'do' statement causes the repeated execution of a group of statements; three different methods are provided for the control of the repetition. The 'goto' statement causes unconditional transfer of control.

There are two methods for procedure invocation. A procedure can be invoked by a 'call' statement or a function reference. The latter case is of great importance, because it allows a procedure to be called in the midst of the evaluation of an expression and to return a result that is used in the evaluation of the expression.

PL/I has facilities for handling exceptional conditions. Examples of conditions are division by zero, reading of an end-of-file, and use of an array subscript that is out of range. In most cases, condition handling can be left to the built-in mechanisms of PL/I. However, condition handling can be programmed when necessary; for example, a program can be supplied for execution when an end-of-file for a file is read. Furthermore, conditions can be disabled to reduce cost; for example, the subscript range condition can be enabled during debugging of a program but disabled for production.

Input/Output

There are two separate facilities for input/output in PL/I, stream and record input/output. Each is surrounded by its own special purpose features, so the language has many operations in this area.

The facilities for stream input/output are based largely on those of FORTRAN. Stream input/output is intended for dealing with hard copy, such as cards, listings, and print-outs, rather than permanent storage. Statements with the 'list' option can be used for input/output with a minimum programming effort. Statements with the 'edit' option can be used to lay out and label values in an elaborate way on the pages of a print-out.

The facilities for record input/output are based largely on those of COBOL. Record input/output can be used for both hard copy input/output and for communication with permanent storage. The statements for record input/output are much simpler than those for stream input/output, so formatting is left to other language features, such as pictured variables.

APPLICATIONS OF PL/I

The use of PL/I in the three major application areas, scientific, business, and system programming is considered here.

Scientific Programming

Many of the features of PL/I are derived from two earlier languages for scientific programming, FORTRAN and Algol; in fact, the development of PL/I began with an effort to develop a new version of FORTRAN. Therefore, many of the features of PL/I may be familiar to the programmer with a background in scientific programming.

Most scientific programs can be written using a small subset of PL/I. Such programs are more readable and compact than the corresponding FORTRAN programs would be; and, in Multics, they are compiled into programs of comparable efficiency. The following guidelines specify a scientific subset of PL/I:

1. Data. Use only the following data types for variables:

- fixed binary(n)
- float binary(n)
- complex float binary(n)
- character(n)
- bit(1)

This eliminates a large part of the language: decimal numbers, fractional fixed-point numbers, picture variables, and all of the noncomputational variables.

2. Aggregates. Use arrays but not structures. This eliminates the declaration and resolution of structure names.

3. Storage Management. Use only the following storage classes:

- automatic
- static
- parameter

Since the 'parameter' attribute takes care of itself, the choice is really the simple one between the default 'automatic' and the occasionally useful 'static'. This eliminates all programmed storage management.

4. Expressions. Use only scalar expressions, and use only simple or subscripted references. This eliminates many unfamiliar features of PL/I.

5. Operations. Use only the following operations:

- arithmetic operations
- mathematical operations
- array operations

This eliminates more than half of the operations.

6. Condition Handling. Allow conditions to be handled by default except for the 'endfile' condition. This eliminates most uses of the 'on' statement.

7. Input/Output. Use stream input/output for most input/output. Use the 'list' option for easy programming or the 'edit' option when the format and layout is elaborate. Use record input/output only for permanent storage of large arrays as Multics files.

The subset of PL/I just described is a language not much larger than FORTRAN IV. The advantage of PL/I is that a particular application program can grow in complexity without exceeding the limits of the full PL/I language. If features not included in the subset are needed for increasing the efficiency, reliability, capacity, or interactiveness of a program, the necessary features are available as part of full PL/I.

Business Programming

PL/I is very different from COBOL, especially in its program structure, computational forms, flow of control, and procedure invocation. Furthermore, certain facilities of COBOL have no counterpart in PL/I; for example, the SORT and MERGE verbs and the report writer. Therefore, PL/I is not easily accepted as a programming language for business applications.

Nevertheless, PL/I was designed to accommodate business programming. It includes generalizations of some of the most successful language features of COBOL, notably picture clauses, structured records, and record input/output. Furthermore, many of its facilities, unfamiliar though they may be, are well-suited to data processing. The success of the method of programming called structured programming has given new impetus to the use of PL/I for business programming because PL/I is well-suited for structured programming and COBOL is not.

Many of the features of PL/I are not required for business programming. The following guidelines specify a business subset of PL/I:

1. Data. Use only the following data types for variables:

- fixed bin(n)
- fixed decimal(m,n)
- character(n) varying
- character(n) nonvarying
- bit(1)
- picture"ps"

This eliminates complex and floating numbers and all of the noncomputational variables.

2. Aggregates. Use both arrays and structures.
3. Storage Management. Use only the following storage classes:

- automatic
- static
- parameter

Since the 'parameter' attribute takes care of itself, the choice is really the simple one between the default 'automatic' and the occasionally useful 'static'. This eliminates all programmed storage management.

4. Operations. Use only the following operations:

- arithmetic operations
- concatenation operator and substring functions
- system counter functions ('lineno', 'pageno', 'time', and 'date')

The arithmetic operations alone are sufficient except where the manipulation of text is necessary.

5. Condition Handling. Allow conditions to be handled by default except for the following:

- endfile
- endpage
- key
- undefinedfile

This eliminates most uses of the 'on' statement.

6. Input/Output. Use record input/output for all input/output. This eliminates the complicated facilities for stream input/output.

This subset is intended to be a guide for the study of PL/I. It is not intended to restrict the use of PL/I; for example, if a programmer already is familiar with stream input/output he should certainly use it where it is convenient.

System Programming

The most impressive application of PL/I is system programming. PL/I is the only widely available language that permits efficient system programming in a high-level language context. Examples of the use of PL/I as a system programming language are close at hand: both the Multics system and the Multics PL/I compiler are written in PL/I.

In the following paragraphs, the features of PL/I that are important for system programming are described.

DATA

Any data structure can be described by an appropriate PL/I variable. A table of data can thus be laid out in a natural and convenient way. The packing of data within a table is completely under the control of the programmer; consequently, he is able to define any pattern of bits. He is able to control the size of critical data bases, and can describe such system-dependent data as page tables, interrupt vectors, input/output channel control words, or machine instructions in object programs.

PL/I based variables are valuable in system programming. They provide the programmer with a completely dynamic storage allocation, a powerful form of list processing, and a mechanism for accessing a data base that occupies any given storage location. Through the use of explicitly allocated based structure variables, the PL/I programmer can dynamically create lists, rings, trees, directed graphs, and so on, whose component nodes are based structure variables containing pointers to other nodes.

PROGRAM STRUCTURE

The structure of a PL/I program closely parallels the modular structure of large systems. A PL/I program can consist of several external procedures that call each other, communicating information through argument lists and external variables. An external variable is declared within each procedure that wishes to use it, and all such declarations are equivalent. The variable exists within the address space of the program but is not owned by any procedure of the program.

The system designer can precisely define a module's interface by actually writing PL/I declarations of external variables and procedure entries. By appropriate use of libraries of these declarations and the %include macro, the project managers can ensure that all the modules use the same declarations of their shared data.

EFFICIENCY

The Multics PL/I compiler was designed to compile PL/I programs into efficient code. In nearly all cases, storage types can be determined by the compiler. Therefore, most references, conversions, and operations can be compiled as optimized in-line code with very little run-time testing.

The handling of string data is a good example of a language feature that is designed for efficient implementation. A PL/I character string value is a sequence of characters whose length is unlimited and is determined during program execution. In contrast, a string variable is allocated with a fixed length. The language must reconcile this difference, and the use of string data is somewhat more difficult than it would be if string variables had unlimited length. However, the PL/I treatment of character strings allows the compiler to produce efficient, noninterpretive object code for operations on character strings, by using the string processing hardware.

The storage management mechanisms of the language can be implemented efficiently. Static storage can be allocated before execution. Automatic storage requires a stack, which can be implemented by means of a base register at minimal cost. Based storage requires a pair of space management routines that are used when the program calls for allocation or freeing of storage; these routines can avoid garbage collection by using threaded lists to keep track of storage.

The parameter passing mechanism of PL/I was designed to permit compilation of reasonably efficient object code. The calling program always has a declaration of the parameters of the called procedure, even when the called procedure is in a separately compiled part of the program. Therefore arguments can be passed to parameters without any interpretive code.

A major cost of program execution is procedure invocation and the dynamic binding of external procedures and external variables. However, these features are essential to good program organization, and they should not be avoided. The PL/I compiler takes special measures to reduce the cost of procedure invocations, and Multics permits the programmer to fully bind his program when it is ready for production use.

PROGRAM VALIDITY

A valid program is one which makes sense according to the definition of PL/I. A program is invalid if the definition of PL/I does not define the result of executing the program. Validity has little to do with whether a program does what the programmer wants it to do. A valid program can certainly yield incorrect results; and, sometimes, an invalid program can yield correct results. However, such an invalid program may not produce consistent results in future versions of the implementation.

Examples of Invalid Programs

The simplest case of an invalid program occurs when the rules of syntax are violated. Consider the following example:

```
P:  proc;
    dcl sysprint file;
    put list("Hello")
    end;
```

This example is not a valid program because the statement on the third line does not end with a semicolon.

Most of the syntactic rules of PL/I are easy to learn; in fact, a knowledge of syntax comes automatically from reading examples of valid programs. Furthermore, when a question of syntax does arise, it can be answered in a formal and almost mechanical way by the syntactic formulae that appear in the PL/I Language Specification.

It is possible for a sequence of characters to be a syntactically valid program but not a semantically valid program. Consider, for example, the syntactically valid program:

```
Q:  proc;
    dcl sysprint file;
    dcl x float;
    put list(x**3);
    end;
```

In this program, the variable designated by 'x' is not set (assigned a value) before it is used (evaluated). The definition of PL/I says that the value of a variable is undefined until it is set, either by explicit initialization or by assignment. Therefore the value output by the 'put' statement is not defined and it follows that the program is not valid.

The rules of PL/I could have been designed so that any syntactically valid program would be completely valid. The undefined cases are not oversights; they were deliberately included to allow more efficient and reasonable interpretation of programs. Consider the example just given, the program 'Q'. If PL/I initialized all variables (say to zero), then 'x' would have a value when it was used in the 'put' statement and the program 'Q' would be valid. But in most cases, when a variable is used before it is set, the program is incorrect, regardless of whether the definition makes it valid or not. Thus automatic initialization would be a wasted operation. Certainly the example program seems to be incorrect, because a program that always prints the same value (such as zero) is not useful.

A program can be invalid when it is used with improper input. Consider, for example, the following syntactically valid program:

```
R:  proc;
    dcl (sysin,sysprint) file;
    dcl x fixed dec(3);
    get list(x);
    put list(x**2);
    end;
```

This program is invalid unless it is used with input values whose magnitudes are less than 1000. If a larger value is supplied as input, then the value will not fit in storage and the value of 'x' is not defined.

Observe, once again, that PL/I could have been designed so that the program just given would always be valid. The size of every value assigned to a variable could be checked before assignment, and a specific action could be taken if the value was too large. However, such a check would be costly; and since most assignments are made within a program where values can be controlled, it would be wasteful. Instead of checking every value, PL/I allows the programmer to explicitly call for a check where it is needed. The example just given can be made valid for all input values as follows:

```
R:  proc;
    dcl (sysin,sysprint) file;
    dcl x fixed dec(3);
(size): get list(x);
    put list(x**2);
    end;
```

The 'size' condition prefix on the input statement causes the value assigned to 'x' by that statement to be checked. If the value is too large, the system prints an error message and aborts the program, and this is a well-defined action.

The program just given takes a well-defined action for improper input, but the action is drastic. The following program is designed to recover from improper input:

```
R:  proc;
    dcl (sysin,sysprint) file;
    dcl size cond;
    dcl x fixed dec(3);
    on size
        begin;
        put list("try again: ");
        put skip;
        goto L;
        end;
(size):
L:   get list(x);
    put list(x**2);
    end;
```

This program responds to improper input by performing a programmed action; specifically, it prints 'try again' and calls for a new input value.

Each of the three versions of the program 'R' are useful under appropriate circumstances. If the programmer can be sure that improper input will not be used, the first version is best because it does not entail the extra cost of the size check. If the programmer considers improper input to be a rare and unimportant occurrence, then the second version is best because it is safe but simple. If the programmer considers improper input to be a normal occurrence in the use of the program, the last version is best because it recovers.

Interpretation of Invalid Programs

The interpretation of an invalid program is not defined for PL/I; however, something happens when an invalid program is compiled and executed. The following cases apply:

- A program that is syntactically invalid is usually detected by the compiler. The compiler prints a diagnostic message and usually declines to produce the object segment.
- Programs that are invalid for nonsyntactic reasons are often not detected by the PL/I system, because the cost of checking for such invalid programs is too great. For example, when a variable is used before it is set, some particular value is used; and unless the value happens to make something go wrong elsewhere, program execution proceeds.

The detection of constructs that are invalid for nonsyntactic reasons is a major part of the debugging of a program.

SUGGESTIONS FOR THE STUDY OF PL/I

The following paragraphs describe various publications that are designed for the study of PL/I. The description begins with a list of three introductory texts, continues with remarks on this manual, and concludes with notes on related Honeywell publications.

Introductory Texts

An introductory text on PL/I provides examples and a framework for further study. Three useful texts are:

1. PL/I Programming Primer, by Gerald M. Weinberg, McGraw Hill, 278 pages. This short book provides a smooth and efficient introduction to PL/I. It keeps strictly to the subject of elementary PL/I, and can be read in a short time. The use of a single programming problem throughout most of the book provides continuity; and the example program can later be run as a Multics PL/I program.

2. PL/I for Scientific Programmers, by C. T. Fike, Prentice-Hall, 241 pages. This book is recommended for the experienced FORTRAN programmer. Most chapters end with a section called "PL/I and FORTRAN" and thus emphasize the relationship between the two languages. The book is organized according to the features of the language rather than having a narrative form; therefore, it cannot be read as quickly as Weinberg's book.
3. An Introduction to Programming, by Richard Conway and David Gries, Winthrop Publishers, Cambridge, Massachusetts, 460 pages. This book is a complete course in high-level programming that is based on PL/I. It is too long for a quick reading, but it provides good background reading because its examples are all in PL/I. Included are discussions of top-down program development, confirmation of program correctness, recursive programming, scientific programming, and file processing.

There are many other introductory texts for PL/I, but most of them have the disadvantage that they go into details of specific computer hardware and specific operating systems, topics that do not contribute to the study of Multics PL/I.

Contents of this Manual

A variety of tutorial techniques are used in this manual. Examples are given in abundance and are often designed to cover all possible cases of a feature. Principles of design of PL/I are given to provide a framework for the details of the language. Guidelines for efficient and clear programming are given when, as is often the case, the language allows more than one way of programming a particular operation. Finally, repetition is freely used to avoid the use of cross references and to emphasize the features of the language that are most important.

Many of the examples in this manual are complete programs. The use of complete programs has several advantages. First, such examples provide a guide for the details of programming style by showing how programs should be laid out for readability, how abbreviations should be used, and, generally, how statements should be put together to form a program. Further, an example that is a complete program does not require a discussion of the context of the example; it is, by definition, complete. Finally, the complete example programs can be used by the reader to experiment with the execution of PL/I programs in the Multics system.

However, most of the examples are not realistic applications of PL/I. A realistic PL/I program, even a small one, requires several pages of background and explanation, and such a discussion is beyond the scope of this manual. The examples in this manual are designed to show how certain statements are executed, not how they should be used to solve problems. Sometimes the absence of realism is obvious, as with an example program that does nothing but read two numbers and print out their sum. In other cases, the examples are complicated; nevertheless, these examples are usually contrived in order to illustrate an important feature of PL/I rather than to solve a real problem.

PL/I provides many ways to program the solution of any given problem. Usually it is not sufficient to write a program that produces the correct results; in addition, the program must be reasonably efficient and fairly readable. The guidelines given in this manual are designed to assist programmers in choosing among the many options offered on PL/I. When a reader has guidelines that are better for his own purposes than those given in this manual, he should ignore those given here.

In some places in this manual, guidelines are given under a special heading, such as "Guidelines for Arithmetic Data Types"; in other places, guidelines are mentioned as part of the definition of a feature. Usually, a guideline is distinguished by the use of the word "should" instead of "must"; for example, "when a storage unit is used for an exact integer, it should be 'fixed binary'."

*

CONTENTS (cont)

Page

Floating-Point Pictures	14-38
Character Pictures	14-38
Control Format Items	14-39
'x' Format Item	14-39
'column' Format Item	14-40
'skip' Format Item	14-40
'line' Format Item	14-40
'page' Format Item	14-40
Format Lists	14-41
Remote Format Items	14-41
Iterated Format Lists	14-41
End-Around Repetitions	14-41
Guidelines for Edit-Directed Input/Output	14-42
String Option	14-43
Special Features	14-44
'read' Statement	14-44
'write' Statement	14-44
Conditions for Stream Input/Output	14-44.1
'conversion' Condition	14-45
'endfile' Condition	14-45
'endpage' Condition	14-46
'name' Condition	14-46
'transmit' Condition	14-46
'undefinedfile' Condition	14-47

Section XV

Record Input/Output	15-1
Record Data Sets	15-2
Organization of Record Data Sets	15-2
Multics Files	15-2
Record Files	15-3
File-State Blocks	15-3
File References	15-4
File Attachment	15-5
Attaching a Switch	15-5
Attach Description	15-5
Opening a Switch	15-6
Opening a File	15-6
Record Input/Output Operations	15-7
Opening and Closing Files	15-8
'open' Statement	15-8
File Descriptions	15-9
'close' Statement	15-10
Keyed Input/Output Operations	15-10
Keyed 'write' Statement	15-11
Keyed 'read' Statement	15-11
Keyed 'delete' Statement	15-12
Keyed 'rewrite' Statement	15-12
Example of Keyed Input/Output	15-13
Sequential Input/Output Operations	15-13
Sequential 'write' Statement	15-14
Sequential 'read' Statement	15-14
Sequential 'delete' Statement	15-14
Sequential 'rewrite' Statement	15-15
Example of Sequential Input/Output	15-15
Based Input/Output Operations	15-16
Based Input	15-16
Examples of Based Input	15-17
Based Output	15-20
Omission of the 'from' Option	15-20
'locate' Statement	15-20
Special Features	15-21

CONTENTS (cont)

	Page	
Conditions for Record Input/Output	15-21	*
'endfile' Condition	15-22	
'key' Condition	15-23	
'record' Condition	15-23	
'transmit' Condition	15-24	
'undefinedfile' Condition	15-24	
 Section XVI		
PL/I in the Multics System	16-1	
Storage System	16-1	
Names	16-1	
Component Names	16-2	
Entry Point Names	16-2	
Reference Names	16-2	
Multics PL/I Compiler	16-3	
Entry Names	16-3	
Object Segment	16-4	
Listing Segment	16-4	
Linking	16-5	
Search Mechanism	16-5	
Hidden Dangers of Dynamic Linking	16-6	
Binding	16-7	
Initialization and Allocation of Variables	16-8	
Attaching Files	16-9	
I/O Switch	16-10	
Standard Switches	16-10	
io call Command	16-10	
I/O Modules	16-11	
vfile I/O Module	16-11	
tty I/O Module	16-11	
syn I/O Module	16-12	
record_stream I/O Module	16-12	
Running A PL/I Program in Multics	16-13	
Entering an External Procedure	16-14	
Compiling an External Procedure	16-14	
pl1 Command	16-14	
Executing a Program	16-15	
Program Termination	16-16	
Debugging a Program	16-16	
probe Command	16-17	
trace Command	16-18	
Measuring a Program's Performances	16-18	
Example of Running PL/I	16-19	
Entering the Example	16-19	
Compiling the Example	16-20	
Executing the Program	16-21	
Program Listing	16-21	
Source Listing	16-22	
Symbol Listing	16-22	
Error Listing	16-23	
Map Listing	16-23	
 Appendix A		
Guide to PL/I Statements	A-1	
Preliminary Remarks	A-1	
Syntax Notation	A-1	
Basic Constructs	A-2	
List of Items	A-2	
Choice of Items	A-3	
Optional Items	A-3	
Recursive Diagrams	A-4	
Parts of a Statement	A-4	
Specific Conventions	A-5	

SECTION II

VALUES

The most important feature of PL/I is the great variety of the values that are provided. When a PL/I program performs a calculation, whether business or scientific, it does so in terms of arithmetic values. When a program accepts input from the user or prints out a listing of results, it transmits string values. When a program refers to the storage for its own data or to the code for its own statements, it uses address values. When a program needs to control storage management, it uses an area value. And, when a program needs to treat several values as a single entity, it groups the values into an aggregate value.

In this section, each kind of value is described and the operations that can be applied to the value are summarized. The values are described in an abstract way, without consideration of the way in which they are stored in memory. The purpose of this section is to indicate the computational power of PL/I.

ARITHMETIC VALUES

Standard PL/I does not specify the range of the arithmetic values; instead, it leaves this choice to each implementation of PL/I. Multics PL/I provides a very large range, as follows:

- The greatest magnitude of a Multics PL/I value is about 10^{186} (for decimal floating point) or about 10^{38} (for binary floating point). These values far exceed any measurement encountered in business or scientific applications.
- The smallest magnitude of a value is 10^{-128} (for decimal floating point) and about 10^{-38} (for binary floating point). These values are much smaller than any measurement encountered in practice.
- The precision of arithmetic values is such that two values can differ by as little as one part in 10^{59} (for decimal floating point) or about one part in 10^{19} (for binary floating point). Thus PL/I can supply a very good approximation to any given number.

The arithmetic values just described are called real values. PL/I also supplies complex values. Each complex value is composed of a pair of real values, and thus the range of complex values is determined by the range of the real values.

A number can be expressed exactly as a PL/I arithmetic value only if it can be expressed exactly in decimal positional notation. For example, $73/2$ can be expressed exactly as '36.5' and $13/320$ can be expressed exactly as '0.040621875'; so these are PL/I arithmetic values. On the other hand $1/3$ and the square root of two cannot be expressed exactly this way, and so must be approximated. If a good approximation is acceptable, then Multics PL/I can provide that approximation; otherwise, special techniques must be applied.

PL/I has many operations for the manipulation of arithmetic values. The familiar arithmetic operators are present; in addition, several dozen built-in functions for integer arithmetic, trigonometry, logarithms, and statistics are provided.

STRING VALUES

A string value is a sequence of characters. Standard PL/I does not specify the set of characters that can appear in a string; instead, the choice of characters is left to each implementation of PL/I. The character set used in Multics PL/I is the ASCII character set, given under "The 'collate' Function" in Section IX, "Operations." It is composed of 128 characters, as follows:

- 52 characters that represent the letters of the alphabet in both lowercase and uppercase
- 10 characters that represent the decimal digits
- 18 characters that represent most of the special symbols on a typewriter keyboard
- 14 characters that represent special symbols not found on an ordinary typewriter keyboard, many of which are useful in writing mathematical expressions or non-English text
- 1 character for the blank
- 33 characters that are nonprinting control characters including, for example, horizontal tab, new line, new page, and carriage return

Because this character set includes characters that control the layout of the pages, a printed document of many pages can be composed, stored, and edited as a single PL/I character string value.

The number of characters in a character string value is the length of the value. The maximum length allowed in Multics is very large; it is limited only by the capacity of a Multics segment.

Several operators and built-in functions are provided for the manipulation of string values. String values can be concatenated and a substring of a given string value can be extracted. The equality operators ('=' and '^=') can be applied to strings. A collating order is defined for the character set, so it is possible to apply inequality operators (such as '<') to strings. Other functions are available for use in such advanced applications as command interpretation and compiler construction.

A special kind of string value is recognized; namely, a string value composed only of the bits '0' and '1'. Such a string value is a bit-string value. When a bit-string value contains a single bit, it is a Boolean value; and it represents "true" or "false" depending on whether the single bit is '1' or '0'. By extension, a bit-string value of length greater than one can be treated as a sequence of true/false indicators. The familiar Boolean operators are provided for use with the bit-string values.

Any arithmetic value can be converted to a string value; the result is a conventional representation of the arithmetic value. Conversely, a string can be converted to an arithmetic value, but only if the string value is a valid representation of an arithmetic value. These conversions are essential because a value must be expressed as an arithmetic value if an arithmetic operation is applied to it, but must be expressed as a string value before it can appear in the output stream. In addition to providing functions that perform these conversions in a straightforward way, PL/I has pictured character-string variables, which allow numeric values to be maintained as character-string values in an automatic and user-controlled way.

ADDRESS VALUES

Every PL/I statement in a program has a unique address, and every data storage unit also has a unique address. It is common in computing to think of an address as an integer and thus endow it with arithmetic properties; but as a value in PL/I, an address has no other property than its association with a particular program statement or storage unit. For example, the "next" address is not defined for any address, so addresses are not even ordered.

The manipulation of address values in PL/I is deliberately limited. An address value can be assigned to a variable and can be produced by a reference to a constant, a variable or a function. The operators '=' and '^=' can be used to determine whether or not two address values are identical. The only conversion that can be applied to address values is that between a pointer value and an offset value.

There are three kinds of address values: statement, locator, and file values; and descriptions of these follow.

Statement Values

A statement value designates a statement in a program. The value is classified as a label, entry, or format value according to the following rules:

- A label value designates a statement to which control can be transferred by a 'goto' statement. Any statement except a 'procedure', 'entry', or 'format' statement can be designated by a label value.
- An entry value designates a statement to which control can be transferred by a 'call' statement or a function reference. There are two such statements, the 'procedure' statement and the 'entry' statement.
- A format value designates a statement that can supply a format list to a stream input/output statement. There is one such statement, the 'format' statement.

Locator Values

A locator value designates a storage unit for program data. There are two types of locator values, as follows:

- A pointer value is used by itself to access a storage unit.
- An offset value is used in conjunction with an area name to access a storage unit.

The pointer value is similar to the absolute machine address used in assembly language programming, and the offset value is similar to a relative or based machine address. Conversion between a pointer value and an offset value can be performed relative to a given area value.

File Values

In Multics PL/I, a file value designates a collection of stored values called a file-state block. The values in the file-state block record the status of a data set that is currently being used for input/output. The file value is a more specialized kind of address value than either the program address value or the locator value.

AREA VALUES

An area value is an ordered set of PL/I values. The entire set of values can be assigned to an area variable, passed as a procedure argument, returned as a procedure result, or transmitted as input or output. However, the principal use of an area value is as part of a specialized mechanism for the efficient management of storage. The most important operations on an area are the allocation and freeing of storage for new values within the area, and the details of these operations are not relevant to this description of values. The area value plays a role in the definition of the 'offset' value, already mentioned under "Locator Values".

AGGREGATE VALUES

An aggregate value is an ordered set of PL/I values. Because it is a set of values, it provides a way of handling a collection of values as a single computational entity. Because it is ordered, it has a first component, a second component, and so on. And because it can contain any PL/I values, it can have another aggregate as a component. Thus an aggregate value can be used to collect values together and arrange them in a hierarchical manner, with aggregates within aggregates.

There are two kinds of aggregate, the array and the structure. An array must contain values that are all of the same kind, whereas a structure is not restricted in this way. Aside from this, the two kinds of aggregate differ in the way they are stored and referenced.

Most of the operations of PL/I can be applied to aggregate values. Each operation is not individually redefined for this purpose; instead, a general rule is applied: the ith component of the result of the operation is produced by applying the operation to the ith component of each of the given operands. Thus the aggregate operation is defined in terms of the conventional, nonaggregate, operation.

CLASSIFICATION OF VALUES

The previous paragraphs have traced a progression from the mathematical values of computation to the specialized values required for efficient programming. The arithmetic values are those universally accepted for use in calculation. The string values are used for objects of universal importance (printed pages); but their precise and restricted definition is characteristic of computer programming rather than conventional usage. The address values are meaningful only in relation to the program statements or stored data, and so are creatures of computer programming. The area values represent a further descent into the special techniques of efficient programming. The aggregate values appear to swing back toward the world of mathematics; but their usage is oriented toward programming rather than mathematics.

The following list is a hierarchy for the PL/I values described in this section. Some useful supplementary terminology has been introduced.

- scalar: any PL/I value that is not an aggregate
- computational: a value of interest beyond programming
 - arithmetic: a number
 - string: a sequence of ASCII characters
 - character string: an unrestricted string
 - bit string: a string of zero's and one's
 - pictured character-string: a specially restricted string
 - non-computational: a value of interest only for programming
 - address: a value that designates a statement or a storage unit
 - statement: the address of a statement
 - label: the destination of a transfer
 - entry: a 'procedure' or 'entry' statement
 - format: a 'format' statement
 - data: the address of data
 - locator: the address of storage for a value
 - pointer: an "absolute" address
 - offset: a "relative" address based in an area
 - file: the address of a file-state block
- area: values gathered together by storage management
- aggregate: an ordered set of values
 - array: components must have the same data type
 - structure: components may differ in data type

CONTENTS (cont)

Page

Allocate	A-5
Assign	A-6
Begin	A-6
Call	A-6
Close	A-7
Declare	A-7
Default	A-8
Delete	A-8
Do	A-9
End	A-10
Entry	A-10
Format	A-11
Free	A-12
Get	A-13
Goto	A-14
If	A-15
Locate	A-15
Null	A-16
On	A-16
Open	A-17
Procedure	A-18
Put	A-19
Read	A-20
Return	A-21
Revert	A-21
Rewrite	A-21
Signal	A-22
Stop	A-22
Write	A-22

Appendix B

New Features	B-1
'signed' and 'unsigned' Attributes	B-1
'9-bit' and '4-bit' Decimal	B-2

TABLES

Table 3-1	Boundary and Length for Scalar Variables	3-67
---------------------	--	------

Index	i-1
-----------------	-----

SECTION III

VALUE STORAGE

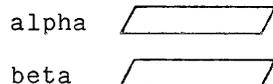
A principal objective of the designers of PL/I was to provide a language in which efficient use of the computer hardware was possible. Therefore, PL/I allows a programmer to give a detailed description of the storage used for each value that is generated during program execution. Since there are many ways to store a value in computer memory, there are many types of value descriptions in PL/I; in fact, the size and complexity of PL/I is largely a consequence of the many ways in which value storage can be described.

As this section continues, it gives a brief description of storage units, which are conceptual models for the storage used to hold PL/I values during program execution. After that, the section gives a long description of the storage types, which are used to describe the kinds of values accommodated by a given storage unit. Most of the description of the storage types follows the same outline as Section II, "Values," dealing with arithmetic, string, address, area, and aggregate values; however, a new concept, alignment, is described at the end of the section.

STORAGE UNITS

Whenever a program refers to a value in any way, that value resides in a storage unit. A storage unit is a conceptual model for the storage of any PL/I value used in executing a program. When a constant appears in a program, it refers to a storage unit that contains an unchanging value. When a variable name appears, it refers to a storage unit that is used to store a computed result for later use. Even a function reference or an operator expression designates a storage unit in which its result is stored, briefly, until that value is used elsewhere. Relative to the actual implementation of Multics PL/I, the storage unit is a simplification. In the implementation, a constant value is not actually stored in the same way as a variable value but is instead a part of the object code. Further, an intermediate result, generated during the evaluation of an expression, is not stored in the same way as a program variable but may reside in a high speed register.

Suppose that the variable names 'alpha' and 'beta' are used in a PL/I program; then a portion of data storage can be diagrammed as follows:



This diagram contains two storage units. Each storage unit consists of a designator, which is the name of a variable, and a box, which can hold a value. Suppose, next, that the following assignment statements are executed:

```
alpha = 3.8;
beta = -2*alpha;
```

Immediately after these assignments, the storage units become:

```
alpha  +3.8
beta   -7.6
```

Each storage unit now has a contents as well as a designator and a box.

STORAGE TYPES

It is possible to design a programming language in which each storage unit can hold any type of value. Some of the interactive languages for the solution of simple problems by nonprogrammers are designed in this way. In such a language, the same variable can accommodate any kind of number or, for that matter, an array of 50 numbers; and if the language has string values or address values, then the variable can also accommodate them. Such a language is easy to learn, but its programs are executed much less efficiently than they could be.

A principal requirement for the efficient execution of a program is the restriction of the kinds of values that can be assigned to a given storage unit. In PL/I, this restriction is applied by associating a storage type with each storage unit. The storage type gives three kinds of information, as follows:

- The data type describes the range and representation of storage for a single datum, such as a number or a character string or a program address.
- The aggregate type describes the way storage for a collection of values is arranged in an array or a structure.
- The alignment type describes the way the storage is laid out in hardware memory and thus determines the memory required and the ease of access.

PL/I provides for efficiency in two ways. First, it requires a storage type so that the range of the storage unit is known when the program is compiled. Second, it provides many different storage types so the programmer can choose the representation best suited to the problem.

Two introductory examples of the interpretation of storage types follow. The examples depend on rules that are given later in this section, when the various data types, aggregate types, and alignment types are described.

As the basis for the first example, consider the following 'declare' statement:

```
dcl gamma fixed bin(8);
```

This statement gives the storage type for the variable named 'gamma'; it does so by means of the scale attribute 'fixed' and the precision attribute '(8)'. The statement uses various abbreviations and defaults for the storage type, and without them the statement would be:

```
dcl gamma real fixed binary precision(8,0) aligned;
```

According to this declaration, the storage type for 'gamma' is as follows:

- The data type is 'real fixed binary precision(8,0)'. This means that the contents of the storage unit named 'gamma' must be a number that can be expressed as a signed eight-digit binary integer.
- The aggregate type is scalar since no aggregate type is explicitly given in the 'declare' statement. This means that the storage unit accommodates a single number, not an array or a structure of numbers.
- The alignment type is 'aligned'. In this case, a full word must be allocated for the variable, so that access to the variable is efficient.

The details of the data type for this example are discussed later in this section under "Arithmetic Storage".

Suppose the following assignment statement is executed:

```
gamma = -15;
```

The following is a diagram of the storage unit for 'gamma' as just declared:

```
gamma      fixed bin(8)  
           ┌───────────┐  
           │  -15      │  
           └───────────┘
```

The storage unit now has the storage type written above it as well as a designator, a box, and a contents. Observe that the contents is a value that satisfies the restriction imposed by the storage type.

As a second example of the declaration of the storage type, consider the following:

```
dcl 01 customer aligned,  
    02 name char(18),  
    02 code(2) fixed dec(4);
```

This statement gives the storage type for the variable named 'customer'; it does so by means of attributes and level numbers. The storage type, by itself, is:

```
01 aligned, 02 char(18), 02 dim(2) dec(4)
```

Once again, the statement uses various abbreviations and defaults for the storage type, and without them the statement would be:

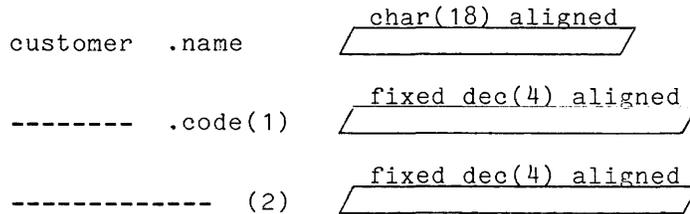
```
dcl 01 customer aligned,  
    02 name character(18) nonvarying aligned,  
    02 code dimension(2) real fixed decimal precision(4,0) aligned;
```

According to this declaration, the variable 'customer' is a sequence of three components, and its storage type is as follows:

- The data type is 'character(18) nonvarying' for the first component and 'real fixed decimal precision(4,0)' for the last two components. This means that the first component accommodates a string of 18 characters and the second and third components each accommodates a signed four-digit decimal integer.
- The aggregate type is '01, 02, 02 dimension(2)'. This means that the variable is a structure with two members, and the first member is a scalar while the second is a one-dimensional array with two elements.
- The alignment type is 'aligned' for all three components. This means that the variable should be laid out in memory to permit efficient access rather than to save space. In Multics, the variable takes nine words; if it had been 'unaligned', it could have been packed into seven words.

The details for this storage type are given later in this section; the purpose of the example is to provide an introduction, not a complete explanation.

The following is a diagram of the storage units for 'customer' as just declared:



In this diagram, the aggregate type is shown by the way the three designators are arranged. The hyphens are used like ditto marks, so that the designators for the storage units are:

```
customer.name
customer.code(1)
customer.code(2)
```

For convenience of discussion, the storage for the entire variable is called a storage unit. Thus one speaks of a structure storage unit that is made up of three component storage units just as one speaks of a structure value made up of three component values.

ARITHMETIC STORAGE

PL/I is designed primarily for operations on arithmetic values. If differences of scaling and precision are ignored, there are eight different ways to represent an arithmetic value in storage. The choice of one of these kinds of storage and the choice of an appropriate precision is a choice between convenience and efficiency. For example, it is more convenient to use decimal numbers throughout, but scientific computations can be performed much more efficiently in binary. For another example, it is more convenient to use a large number of digits for a variable, but it is more efficient to determine exactly how many digits are required and use no more. The selection of the type of storage for an arithmetic variable is an important part of the engineering of a PL/I program.

This discussion begins by defining the arithmetic data types and their corresponding storage units, continues with abbreviations and defaults that allow data types to be written more concisely, gives examples of various arithmetic storage units, and concludes with guidelines for the selection of a data type for a given purpose.

Arithmetic Data Types

The complete data type for an arithmetic storage unit is given as a sequence of four arithmetic attributes. These attributes are the mode, scale, base, and precision, as described in the following paragraphs.

MODE ATTRIBUTE

The mode attribute is one of the following keywords:

real
complex

In scientific applications that make use of the theory of complex numbers, the 'complex' mode may be specified for a variable. In most other applications, the 'real' mode is specified.

A storage unit with the 'real' attribute accommodates a number that can be represented as a signed sequence of digits in which a decimal or binary point appears. This includes all numbers that are used in business applications, system programming, and everyday calculation. It also includes most numbers used in scientific applications.

A storage unit with the 'complex' attribute accommodates a number that can be represented as a pair of real numbers, called the real part and the imaginary part, respectively. Both members of the pair have the same scale, base, and precision, as specified by the other attributes of the data type.

SCALE ATTRIBUTE

The scale attribute is one of the following keywords:

fixed
float

These attribute keywords refer to the decimal or binary point of the number. In a 'fixed' storage unit, the point cannot move, whereas in a 'float' storage unit, the point can be thought of as moving to accommodate a wider range of values in a given number of digits.

A storage unit with the 'fixed' attribute accommodates a value that can be represented as a signed sequence of digits in which a decimal or binary point appears. The point can appear anywhere, but its position is determined when the storage unit is created and remains fixed throughout the existence of the storage unit. For a given 'fixed' storage unit, the number of digits and the position of the point are both specified by the precision attribute, which is described a little later.

A storage unit with the 'float' attribute accommodates a value that can be represented in one of the following forms:

$\underline{m} * (2^{**}\underline{e})$ (if the scale is 'binary')

$\underline{m} * (10^{**}\underline{e})$ (if the scale is 'decimal')

where \underline{m} is the mantissa and \underline{e} is the exponent. The mantissa is a signed sequence of digits in which a point appears. For 'binary' scale, the point appears to the left of the first digit, so that the mantissa is a fraction. For 'decimal' scale, the decimal point appears to the right of the last digit, so that the mantissa is an integer.

The number of digits in the mantissa is determined by the precision attribute, which is described later. The exponent must lie in the range:

$$-128 \leq \underline{e} \leq +127$$

When a value is assigned to a 'decimal' storage unit, there is some freedom in the choice of the mantissa and the exponent. For example, if the mantissa has four decimal digits, then the value 1.4 can be represented in the following ways:

+0014.*(10**-1)
+0140.*(10**-2)
+1400.*(10**-3)

However, a representation is never chosen that discards more low-order digits than necessary, and 1.4 would not be represented as:

+0001.*(10**0)

Thus the variable exponent not only allows a wide range of values but also allows the full use of each digit of the mantissa whenever necessary.

When a value is assigned to a 'binary' storage unit, the mantissa and exponent are always chosen so that the first digit of the mantissa is a one; that is, the mantissa is normalized.

The exponent can be thought of as expressing the number of places the mantissa point should be moved to the right to give the true value of the storage unit. That point of view is the source of the term "floating point" and the keyword 'float'.

BASE ATTRIBUTE

The base attribute is one of the following keywords:

binary
decimal

The attribute keywords refer to the number system that is used in representing the value.

A storage unit with the 'binary' attribute uses binary digits in representing its value. Thus a 'fixed binary' storage unit that has three significant digits followed by a binary point can accommodate the integers from -7 to +7.

A storage unit with the 'decimal' attribute uses decimal digits in representing its value. Thus a 'fixed decimal' storage unit that has three significant digits followed by a decimal point can accommodate the integers from -999 to +999.

PRECISION ATTRIBUTE

The complete precision attribute has one of the following forms:

precision(p,q)

precision(p)

where p and q are the number-of-digits and the scale-factor, respectively. The first form is used when the scale is 'fixed' and the second form is used when the scale is 'float'. The number-of-digits must be given as an unsigned integer constant and the scale-factor must be given as an optionally-signed integer constant. They cannot be given as general expressions because their values must be known to the compiler.

The number-of-digits determines how many digits appear in the storage unit. The number-of-digits must lie within a certain range, and that range depends on the scale and base attributes that appear with the precision attribute. The ranges are:

<u>Scale and Base</u>	<u>Minimum</u>	<u>Maximum</u>
fixed binary	1	71
float binary	1	63
fixed decimal	1	59
float decimal	1	59

This table shows, for example, that the storage type 'real fixed binary precision(70,0)' is valid but 'real fixed decimal precision(70,0)' is not. The ranges are not part of Standard PL/I; they are chosen for each implementation, and the values shown here are those for Multics.

In a 'float' storage unit, the number-of-digits refers to the digits in the mantissa and does not include digits used in the exponent. In the case of a 'float binary' storage unit, the number-of-digits establishes a minimum for the number of mantissa digits. The purpose of this latitude is to permit efficient use of floating-point binary hardware.

The scale-factor determines the number of significant digits to the right of the point in a 'fixed' storage unit. The scale-factor is restricted to the following range:

<u>Scale</u>	<u>Minimum</u>	<u>Maximum</u>
fixed	-128	+127
float	(not applicable)	

A fixed-point storage unit can have filler zeros. These zeros are used to position the point and are not counted in the determination of the number-of-digits. Suppose the precision attribute is 'precision(p,q)'; then three cases must be considered:

- If $p \geq q \geq 0$, then no filler zeros are required because the point is adjacent to one of the significant digits. For example, the storage type 'real fixed decimal precision(5,2)' accommodates any value that can be represented by a sign, three significant digits, a decimal point, and two more significant digits. Thus the value -203.49 or the value 1.2 can be accommodated.
- If $p < q$, then $q-p$ filler zeros are assumed between the point and the first significant digit. For example, the storage type 'real fixed decimal precision(5,7)' accommodates any value that can be represented by a sign, a decimal point, two filler zeros, and five significant digits. Thus the fraction .0044244 is accommodated by this storage type but .0144244 is not.
- If $q < 0$, then $-q$ filler zeros are assumed between the last significant digit and the point. For example, the storage type 'real fixed decimal precision(5,-3)' accommodates any value that can be represented by a sign, five significant digits, three filler zeros, and a decimal point. Thus the integer 55955000 is accommodated by this storage type, but the integer 55955300 is not (and must be approximated).

In practice, most programs use only fixed variables whose precision attribute satisfies $p \geq q \geq 0$; therefore filler zeros are not often used.

ABBREVIATIONS AND DEFAULTS

A typical program uses many variables, and a data type must be given for each variable; therefore, PL/I permits the use of many abbreviations and defaults in the specification of data type attributes.

From the point of view of the PL/I compiler, any selection of abbreviations and defaults can be used, and the selection can differ from one place in the program to another. However, the inconsistent use of abbreviations and defaults makes a program confusing, and a consistent policy should be adopted. This manual provides one such policy: every abbreviation or default of PL/I should be used except for those whose avoidance is explicitly recommended in this manual.

The following abbreviations are defined for the keywords used in arithmetic attributes:

<u>Keyword</u>	<u>Abbreviation</u>
binary	bin
decimal	dec
precision	prec (rarely used)
precision	(omit the entire keyword 'precision' if it immediately follows a mode, scale, or base attribute)
complex	cplx (not recommended)

The abbreviation 'prec' is rarely used because it is customary to write the precision attribute immediately after some other arithmetic attribute and omit the entire 'precision' keyword. For example, 'real fixed decimal precision(5,2)' is abbreviated as 'real fixed dec(5,2)' so that all that is left of the precision attribute is '(5,2)'. The abbreviation 'cplx' is not recommended because it is difficult to remember and impossible to pronounce.

A default attribute is the attribute that is assumed by the PL/I compiler when a required attribute is not given in the program. For example, if a variable is declared 'fixed dec(5,2)', then the compiler treats the variable as if it had been declared 'real fixed dec(5,2)'; it does so because the default attribute for the mode is 'real'. The defaults for the arithmetic data type are:

<u>Omitted Item</u>	<u>Default</u>
mode attribute	real
scale attribute	fixed
base attribute	binary
number-of-digits	17 (for 'fixed binary') 7 (for 'fixed decimal') 27 (for 'float binary') 10 (for 'float decimal')
scale-factor	0 (for 'fixed' scale only)

The table just given shows that the default value for the number-of-digits depends on the scale and base of the data type; thus there are four possible default values. The default values for the number-of-digits are not part of Standard PL/I, and the values given are those for Multics. However, it is understood that any implementation of PL/I should choose a default for 'fixed' values that is appropriate for an index or subscript and a default for 'float' values that is appropriate for most scientific calculations.

Although defaults are specified for the scale and base attributes, these defaults may vary from one implementation of PL/I to another. Therefore, it is recommended that both the scale and base attributes be given explicitly.

The abbreviations and defaults are designed to favor the most commonly used storage types, as shown in the following examples:

<u>Complete Data Type</u>	<u>Recommended Form</u>
real fixed binary precision(17,0)	fixed bin
real fixed binary precision(30,0)	fixed bin(30)
real fixed decimal precision(8,2)	fixed dec(8,2)
real float binary precision(27)	float bin
real float binary precision(60)	float bin(60)
complex float binary precision(27)	complex float bin

Examples of Arithmetic Storage Units

As an example of the use of an arithmetic storage unit, consider the following program:

```
P: proc;
  decl alpha fixed dec(6,2);
  ...
  alpha = -31.253;
  ...
end;
```

The storage unit for 'alpha' can be diagrammed as:

alpha

fixed dec(6,2)
-31.25

The data type is written above the box to indicate the restriction on the value accommodated by the box. Since the scale-factor is two, the storage unit can accommodate only two fractional digits; therefore, the value -31.253 is approximated when it is assigned to the storage unit.

A diagram called a data frame is useful in describing the capacity of a storage unit. A data frame is produced by combining the data type with the box; the result is a diagram that suggests the structure of storage. When a data frame is used for the variable 'alpha' just discussed, then the result is:

alpha

S 9 9 9 9	9 9
-0031	.25

In this diagram, each of the seven boxes holds a single character. The symbol above each box determines the kind of character allowed: 'S' allows a sign, '9' allows a decimal digit, and other symbols are used for other restrictions. This diagram makes it clear that the storage unit accommodates any value that can be represented as a sign followed by four decimal digits followed by a decimal point followed by two decimal digits. It also makes it clear that the last digit of -31.253 must be dropped before assignment to the storage unit can occur.

A data frame provides useful information about the data type as it is applied to a storage unit. The diagram should be viewed as having three components:

- The characters written in the boxes are the contents of the storage unit. They are the only part of the storage unit that can change; but they do not, by themselves, represent the value of the storage. In the example above, the contents is '-003125' and this sequence of characters would not, in itself, be interpreted as the value -31.25.
- The characters in line with the contents but not enclosed in boxes are the interpretation of the storage unit. In the example above, the interpretation is the decimal point '.'. The contents and the interpretation together are a representation of the value of the storage unit, and the representation is always a valid PL/I constant expression. In the current example, the representation is '-0031.25' and this is a valid signed constant and could be used in a program to represent the value -31.25.
- The data frame without contents or interpretation is an indication of the hardware storage required for the storage unit. In the current example, the storage unit has one sign box (one byte in a decimal number) and six decimal-digit boxes (each one byte); therefore the storage unit requires seven bytes of memory in a Multics segment.

The data frames are not a part of the PL/I language itself; they are introduced here as a useful way of describing a storage unit. Since a PL/I program cannot depend on the way in which arithmetic values are represented, the purpose of the data frames is not to show how the components of a value can be operated upon.

FIXED DECIMAL STORAGE UNITS

A 'fixed decimal' storage unit closely approaches everyday notation for a number. The availability of these storage units make it possible to entirely avoid fractional fixed-point binary arithmetic and thus eliminates one of the major problems of computing. Four examples of 'real fixed decimal' storage units follow, each with a different precision attribute. Each example gives the declaration of a variable and then the storage unit that corresponds to the declaration.

```
dcl u1 fixed dec(8,2);      u1  S 9 9 9 9 9 9 . 9 9
```

The complete data type for this storage unit is 'real fixed decimal precision(8,2)'. The data type accommodates any number with magnitude less than one million to the nearest one hundredth. It would be useful for a sum in dollars and cents and, since it has a sign, it could be used for a credit or a debit.

```

      dcl u2 fixed dec(6);           u2      S  9 9 9 9 9 9.

```

The complete data type for this storage unit is 'real fixed decimal precision(6,0)'. It accommodates any integer with magnitude less than one million.

```

      dcl u3 fixed dec(6,9);        u3    S.000  9 9 9 9 9 9

```

The complete data type for this storage unit is 'real fixed decimal precision(6,9)'. The scale-factor is greater than the number-of-digits and therefore filler zeros appear between the decimal point and the first significant digit. The storage unit accommodates a fraction with magnitude less than one thousandth to the nearest billionth.

```

      dcl u4 fixed dec(4,-2);       u4    S  9 9 9 900.

```

The complete data type for this storage unit is 'real fixed decimal precision (4,-2)'. The scale-factor is negative and therefore filler zeros appear between the last significant digit and the decimal point. The storage unit accommodates an integer with magnitude less than one million to the nearest hundred.

FIXED BINARY STORAGE UNITS

A 'fixed binary' storage unit can be declared with the same variety of precisions as were just illustrated for the 'fixed decimal' storage units. In practice, however, a 'fixed binary' storage unit is rarely used with a scale-factor other than zero. Two examples follow.

```

      dcl v1 fixed bin(3);          v1    s 1 1 1.b

```

The complete data type for this storage unit is 'real fixed binary precision(3,0)'. The '1' symbol over a box indicates that it must contain a binary digit. The 'b' at the right end shows that the representation is binary. The storage unit can accommodate, for example, the value -2; the representation for that value is '-010.b'.

```

      dcl v2 fixed bin;            v2    s 1--(17)--  1.b

```

The complete data type is 'real fixed binary precision(17,0)'. The parenthesized 17 means that there are seventeen digit positions in the storage unit. The storage unit accommodates any integer whose magnitude is less than 2**17 (which is 131072). It is the recommended storage unit for most indexes and subscripts of nonstring data.

FLOAT BINARY STORAGE UNITS

A 'float binary' storage unit is compatible with the floating-point binary hardware operations that are part of many computers. Scientific applications use 'float binary' storage units for physical variables. As already stated in the description of the scale attribute, a 'float binary' storage unit accommodates a value of the form:

$$m * (2^{**}e)$$

where m is the mantissa and e is the exponent. The range of a 'float binary' storage unit is determined primarily by the fact that its exponent lies in the range -128 through +127. In Multics, any 'float binary' storage unit can accommodate values whose magnitudes are in the range 10^{**-38} to 10^{**+38} . Three examples of 'float binary' storage units are given here.

```
dcl x1 float bin(8);      x1   $\frac{s}{\square}.\frac{1}{\square}--(8+)--\frac{1}{\square}e\frac{exp}{\square}b$ 
```

The complete data type for this storage unit is 'real float binary precision(8)'. The mantissa is a signed fraction with eight or more digits. When the representation is written, the exponent is written as an optionally-signed decimal integer, but in the hardware it is represented in seven bits. The 'b' at the end of the representation applies only to the mantissa. The storage unit can accommodate, for example, the value 4.5; the representation used for that value is '+.10010000e3b'.

```
dcl x2 float bin;       x2   $\frac{s}{\square}.\frac{1}{\square}--(27+)--\frac{1}{\square}e\frac{exp}{\square}b$ 
```

The complete data type for this storage unit is 'real float binary precision(27)'. In Multics, the mantissa has 27 digits. This data type (without an explicit precision attribute) is used for most scientific variables.

```
dcl x3 float bin(28);  x3   $\frac{s}{\square}.\frac{1}{\square}--(28+)--\frac{1}{\square}e\frac{exp}{\square}b$ 
```

The complete data type for this storage unit is 'real float binary precision(28)'.

FLOAT DECIMAL STORAGE UNITS

A 'float decimal' storage unit is used only under exceptional circumstances. It can be used to get more precision than is available from 'float binary' since 59 decimal digits of precision is equivalent to about 196 binary digits of precision. It can also be used to avoid binary representation, but the operations on 'float decimal' values are so much more expensive than those on 'float binary' values that this course is seldom followed. A 'float decimal' storage unit accommodates a value of the form:

$$m * (10^{**}e)$$

where m is the mantissa and e is the exponent. Any 'float decimal' storage unit can accommodate values whose magnitudes are in the range 10^{**127} to 10^{**-69} . One example is given here.

```
dcl y1 float dec(59);    y1  $\overline{\overline{s}}^9 \overline{\overline{--}}(59) \overline{\overline{--}}^9 . e \overline{\overline{exp}}$ 
```

The complete storage type for this storage unit is 'real float decimal precision(59)'. The mantissa is a signed decimal integer with 59 digits. This storage unit has greater precision and range than any other in PL/I. It occupies 16 words of a Multics segment.

COMPLEX STORAGE UNITS

A 'complex' storage unit can have any scale, base, or precision attributes. In practice, however, it is always used with the 'float binary' attributes. Since a complex number is a pair of real numbers, a 'complex' storage unit is formed by placing two 'real' storage units together and marking the second as the imaginary part by suffixing an 'i' to it. An example is:

```
dcl z1 complex float bin;
```

```
z1  $\overline{\overline{s}}^1 \overline{\overline{.}} \overline{\overline{1}} \overline{\overline{--}}(27+) \overline{\overline{--}}^1 e \overline{\overline{exp}} b \overline{\overline{s}}^1 \overline{\overline{.}} \overline{\overline{1}} \overline{\overline{--}}(27+) \overline{\overline{--}}^1 e \overline{\overline{exp}} bi$ 
```

Everything through the first 'b' represents the real part of the storage unit and the remainder is the imaginary part.

Guidelines for Using Arithmetic Data Types

The choice of data types for the numeric variables is a major part of the design of a program. The first choice that has to be made is between the arithmetic storage types, just described, and the pictured storage types, described later in this section. The arithmetic storage types can be operated on efficiently but require conversions when input/output is performed; they are appropriate when calculations are relatively complicated, as in scientific programming, or when the packing of data is important, as in system programming. The pictured storage types are appropriate when a major consideration is the format of input and output, as in business programming.

Once the general decision has been made, other details must be decided. For an arithmetic storage unit, the mode, scale, base, and precision must be selected. These selections effect the range of the input data accepted by the program, the accuracy of the results developed by the program, the convenience of writing and debugging the program, and the cost of executing the program. The questions of range and accuracy depend on the mathematical analysis of the algorithm being programmed. The questions of convenience and cost depend on the human and mechanical factors of the programming system. Guidelines for the choice of the attributes of an arithmetic storage unit are given here.

CHOICE OF MODE

The choice of mode is entirely a question of range. Within a given scientific program, the requirement for a 'complex' mode attribute is determined by the mathematical formulation of the given program. Except for scientific applications, the mode is usually 'real'; and since the default mode is 'real', most programs never make explicit use of a mode attribute.

CHOICE OF SCALE AND BASE

The choice of the scale attribute is a choice between the efficiency of the fixed-point operations and the wide range of floating-point arithmetic. The choice of the base attribute is a choice between the efficiency of binary operations and the convenient familiarity of decimal representation. The following rules contribute to the selection of the scale and base for a particular storage unit:

1. When a storage unit is used for an exact integer that remains in a reasonable range, it should be 'fixed binary'. This rule applies to subscripts, indexes, and counters.
2. When a storage unit is used for a noninteger quantity that must be approximated with care, it should be 'fixed decimal'. An example is a dollars-and-cents quantity.
3. When a storage unit is used for a quantity that is inherently approximate, it should be 'float binary'; however, if an exceptional requirement for precision exists, the storage unit should be 'float decimal'. Most physical quantities, such as weight, length, speed, and so on, should be 'float binary' or 'float decimal'.

In some business applications, it is convenient to use the 'decimal' base for all storage units. This avoids the mixing of 'decimal' and 'binary' values; but it may significantly increase a programs execution time.

CHOICE OF PRECISION

The precision attribute gives the number-of-digits for every arithmetic storage unit and gives the scale-factor for 'fixed' storage units. The choice of precision for 'float' variables is usually easy because there is no danger of overflow and a detailed analysis of the accuracy of the calculation is often impossible. The choice of precision for a 'fixed' variable is much more difficult because of the danger of overflow and the possibility that significant digits will be lost. Some form of analysis is usually required for the choice of precision for a 'fixed' variable.

The defaults for the number-of-digits depend on a given implementation of PL/I and can vary from one computer to another. Consider a 'fixed' variable that is used as a counter. If an analysis has shown that the variable will never require more than 10 binary digits, then it should be declared 'fixed(10)'. If, on the other hand, a less precise analysis has determined that a "fairly large" capacity will suffice, then it should be declared 'fixed'. In the first case, the programmer makes a specific assertion about the use of the variable; in the second case, the programmer gives the compiler latitude to select an efficient representation for the given hardware.

The arrays and strings of Multics PL/I are unusually large. An array can have up to 2^{24} elements and a string can have up to 2^{24} bits. When arrays and strings that are very large are used, it is necessary to use 'fixed(24)' variables for subscripts rather than 'fixed' variables since the latter provide only 17 binary digits.

When a large counter is required, the number of digits should be kept less than 35 if possible. Although a 'fixed(35)' variable occupies only one word of Multics memory, operations on the variable tend to get into double precision. Thus 'fixed(30)' would be a better choice.

ORDINARY STRING STORAGE

A string storage unit is either ordinary or pictured. The differences between the two kinds of string storage are more important than their similarities, so they are described separately. Ordinary string storage, described here, is primarily used for text of a general nature, such as column headings, error messages, or complete documents. Pictured string storage, described later, is primarily used for numeric values that are represented as character strings.

Ordinary String Data Types

The complete data type for an ordinary string storage unit is given by a sequence of two string attributes. These attributes are the string-type and the variability, as described in the following paragraphs.

STRING-TYPE ATTRIBUTE

The string-type attribute has one of the following forms:

character(ml)

bit(ml)

where ml is the maximum-length of the string. The maximum length must be an extent: that is, it must have the form:

exp

exp refer (ref)

*

where exp is an expression and ref is a reference. The first form is used in most cases; it must yield a value that can be converted to a 'fixed binary(24)' value. The second two forms are described in Sections VII and XII, "Storage Management" and "Procedure Invocation", respectively.

A storage unit declared 'character(ml)' can accommodate a sequence of n ASCII characters, where n is the value of ml at the time the storage unit is allocated. Similarly, a storage unit declared 'bit(ml)' can accommodate a sequence of n bits, where n is the value of ml at the time the storage unit is allocated. A bit is one of the characters '0' or '1'.

The value of the maximum-length can range from zero to a very large number. If it is zero, then the only value the storage unit can hold is the null string. The maximum-length must be chosen so that the resulting storage unit fits into the Multics segment in which it begins. If the string begins near the end of the segment, this restriction is important; but that happens only when other large storage units are in use. If the string begins at the beginning of a segment and is 'nonvarying', then the maximum-length can be as large as $4*2**18$ for a character string or $36*2**18$ for a bit string.

VARIABILITY ATTRIBUTE

The variability attribute is one of the following keywords:

nonvarying
varying

An ordinary string storage unit with the 'nonvarying' attribute can accommodate a string of only one length: the maximum length specified in the string-type attribute. When a shorter string is assigned to a 'nonvarying' storage unit, blanks or zeros (for a character or bit string, respectively) are added to the right end of the string until it has the required maximum length. An ordinary string unit with the 'varying' attribute can accommodate a string of any length from zero up to and including the maximum length given in the string-type attribute.

ABBREVIATIONS AND DEFAULTS

The following abbreviations are accepted for the keywords used in ordinary string attributes:

<u>Keyword</u>	<u>Abbreviation</u>
character	char
nonvarying	nonvar
varying	var

The use of the abbreviation 'var' for 'varying' is not recommended.

The defaults for the string attributes are:

<u>Omitted Item</u>	<u>Default</u>
variability attribute	nonvarying
maximum-length	1

Some examples of the application of these rules are:

<u>Complete Data Type</u>	<u>Recommended Form</u>
character(n+1) varying	char(n+1) varying
character(1) nonvarying	char(1)
bit(1) nonvarying	bit(1)

The last case is especially useful because a one-bit string is used as a Boolean value in PL/I.

Examples of Ordinary String Storage

As an example of the use of an ordinary string storage unit, consider the following program:

```
P: proc;
    dcl beta char(15);
    ...
    beta = "John Q. Smith";
    ...
end;
```

The storage unit for 'beta' can be diagrammed as:

```
beta  char(15)
      /-----/
      "John Q. Smith "
```

Because the variable is 'nonvarying' by default, two blanks are added at the right end of the assigned character string to make it 15 characters long.

A data frame can be used for an ordinary string storage unit. For the variable 'beta', just discussed, the resulting diagram is:

```
beta  "J o h n Q . S m i t h / / / / /"
      x x x x x x x x x x x x x x x
```

The diagram has 15 boxes, one for each character in the stored string. The 'x' over each box indicates that the box can hold any ASCII character.

CHARACTER STORAGE UNITS

Two examples of 'character' storage units are given here; one is 'nonvarying' and the other is 'varying'.

```
dcl s1 char(80);      s1  " / (30) / "
```

The complete data type for this storage unit is 'character(80) nonvarying'. The storage unit accommodates a character string that is exactly eighty characters long; it could be used, for example, to store the contents of an eighty-column card.

```
dcl s2 char(6) varying;  s2  cnt / " S a m & w 2 / "
```

The complete data type is 'character(6) varying'. The storage unit can accommodate any character string that has from zero to six characters. The storage unit is shown with a value in it; that value is "Sam". The 'cnt' box gives the current length of the string in the storage unit; the current length is filled in each time a new value is assigned to the storage unit. The last three character positions in the storage unit contain characters which, presumably, are left from previous values of the storage unit.

BIT STORAGE UNITS

The 'bit' storage units are defined in the same general way as the 'character' storage units. In practice, however, 'bit' storage units are used in rather specialized ways. Two examples are given here.

```
decl t1 bit;          t1  "1"b
```

The complete data type is 'bit(1) nonvarying'. The storage type is used when a Boolean variable is required. It can have only two values, and these are interpreted as follows:

"1"b means "true"

"0"b means "false"

Such values are used in 'if' statements, loops, and other statements that control program flow.

```
decl t2 bit(5);      t2  "11111"b
```

The complete data type is 'bit(5) nonvarying'. The storage type could be used to hold five flags, each of which represented the truth or falsity of some position. Such bit strings are especially useful in programs for process control.

Guidelines for Using Ordinary String Data Types

A 'character' storage unit can be used in a general way: it is useful for holding a single character, for holding a proper name or a short phrase, or for holding a document of considerable size. Further, both the 'varying' and the 'nonvarying' 'character' storage unit are useful. In contrast, a 'bit' storage unit is specialized: it is usually just one bit in length and it is almost always 'nonvarying'.

As the diagrams given in the examples suggest, the amount of storage required for a string depends on the maximum-length, not on the current length. Thus a 'character(1000) varying' storage unit occupies the same amount of storage (251 words) regardless of whether it contains a string value of length one or 1000.

A 'nonvarying' string is handled somewhat more efficiently than a 'varying' string. Specifically, a 'varying' string requires an extra Multics word to keep the current-length counter in and the code for accessing the storage unit is more complicated. Therefore, when efficiency is the only consideration, a 'nonvarying' string should be used.

Often the choice of the variability attribute is determined by the nature of the data. For example, consider the storage for a textbook that is being edited. An individual line might be kept in a 'nonvarying' string storage unit, since lines are of constant length. On the other hand, an individual word drawn from the text might be stored in a 'varying' storage unit.

PICTURED STRING STORAGE

Pictured-string storage units offer an almost complete alternative to arithmetic and ordinary storage units; that is, many useful programs are written in a natural and convenient way using only pictured storage units. Pictured storage is most often used in business applications, but it can be used very effectively elsewhere.

The pictured storage unit eliminates the use of a special representation for numbers inside the computer; it uses the same representation inside that people use on the outside on cards and listings. The representation is a sequence of characters arranged in a special way: decimal digits, a decimal point, a properly placed sign, and so on. When pictured storage is used, input/output does not involve a conversion between an encoded internal representation and the external text; instead, it simply requires the transmission of a character string.

A pictured storage unit is thought of as having two values. When it is referenced in a context that requires a character string (as in an output statement), then its value is a character string. When it is referenced in a context that requires an arithmetic value (as in a calculation), then its value is an arithmetic value.

The use of a character string to store a numeric value is less efficient than the use of a specially encoded sequence of bits. However, some efficiency can be achieved if the exact form of the character string in a given storage unit is known to the compiler. The purpose of the "picture" that is given in the declaration of a pictured storage unit is to establish the exact form of the string accommodated by the storage unit. As an example, consider the following declaration:

```
dcl omega picture"s999";
```

This declaration asserts that the storage unit designated by 'omega' will accommodate a string of four characters. It also specifies that the first character will be a sign and the remaining characters will be decimal digits. This allows the compiler to conclude that the value of 'omega' can always be interpreted either as a character-string value of data type 'char(4)' or as an arithmetic value of data type 'real fixed decimal(3)'.

Pictured Data Types

The complete data type for a pictured storage unit is a picture attribute followed by an optional mode attribute. These attributes are described in the following paragraphs.

PICTURE ATTRIBUTE

The picture attribute has the form:

```
picture"p"
```

where p is the picture. The picture is a sequence of indicators, each optionally preceded by a replicator. A replicator is a parenthesized, unsigned, decimal integer constant. An example of a picture attribute is:

```
picture"$ (4)$9v.99db"
```

in this example, the picture is:

```
$ (4)$9v.99db
```

The replicator '(4)' means that the following indicator is treated as if it appeared four times. Thus the equivalent picture attribute is:

```
picture"$$$$$9v.99db"
```

The picture contains five different indicators, as follows:

```
$ 9 v . db
```

Each indicator has a special interpretation, and the order in which the indicators are given is significant. A full description of the permitted pictures and their interpretations is given later.

Classification of Indicators

A complete list of the indicators follows. The indicators are classified under functional headings, and these same headings are used later in this section when the individual indicators are fully defined.

<u>Classification</u>	<u>Indicators</u>
no-suppression digit indicator	9
decimal-point indicator	v.
sign indicators	s + - cr db
dollar indicator	\$
zero-suppression digit indicators	z * y
drifting-sign digit indicators	s + -
drifting-dollar digit indicator	\$
insertion-character indicator	. , / b
arithmetic decimal-point indicator	v
fixed-point scale-factor indicator	f(<u>n</u>)
floating-point indicators	e k
floating-point scale-factor indicator	f(<u>n</u>)
nonnumeric indicators	9 a x

In the scale-factor indicators, n is an optionally-signed decimal integer constant. The following distinctions apply:

- The '9' indicator is a no-suppression digit indicator if it appears in a numeric picture; otherwise, it is a nonnumeric indicator and the interpretation is slightly different.

- The 's' indicator is a sign indicator if it is the first 's' in a fixed-point picture, a mantissa picture, or an exponent picture; otherwise, it is a drifting-sign digit indicator. Parallel considerations apply to the classification of '+', '-', and '\$'.
- The 'f(n)' indicator is a floating-point scale-factor indicator if it appears in a picture with a floating-point indicator; otherwise, it is a fixed-point scale-factor indicator.

Observe that eight of the indicators are digit indicators, as follows:

9 z * y s + - \$

where some of these are digit indicators only in certain contexts, as just noted.

Classification of Pictures

It is useful to classify pictures in the following way:

- A picture is non-numeric if it contains either of the following indicators:

a x

Otherwise, the picture is numeric.

- A numeric picture is floating-point if it contains either of the following indicators:

e k

Otherwise, the picture is fixed-point.

Examples are:

<u>Picture Attribute</u>	<u>Classification</u>
"99a999"	nonnumeric
"999999"	numeric
"\$9v.99cr"	numeric
"s9v.999999es999"	floating-point numeric
"sv.999999ks99"	floating-point numeric
"s9v.999999"	fixed-point numeric

MODE ATTRIBUTE

The mode attribute is one of the following keywords:

real
complex

This attribute is the same as the mode attribute for arithmetic storage units. It is used only with a picture attribute that is numeric (that is, that does not have an 'a' or 'x' indicator). When the 'real' attribute is used, the picture remains as it is given; when the 'complex' attribute is used, the picture is doubled to provide for the real and imaginary parts of a complex value. For example, the data type:

picture"s999" complex

describes a storage unit that has eight character positions, the first four of which accommodate a signed, three-digit value for the real part and the second four of which accommodate a signed, three-digit value for the complex part.

ABBREVIATIONS AND DEFAULTS

The following abbreviations are accepted for the keywords used in the pictured string attributes:

<u>Keyword</u>	<u>Abbreviation</u>
picture	pic
complex	cplx (not recommended)

A default convention for the numeric pictured string data type is:

<u>Omitted Item</u>	<u>Default</u>
mode attribute	real

Interpreting Pictured Storage

The interpretation of a storage unit determines the way in which a value is assigned to the storage unit and later fetched. The operations of assignment and fetch do not include data type conversions that are necessary to adjust the value of the storage unit to its context; the conversions are separate operations, discussed in Section IV, "Value Conversion." The operations of assignment and fetch are very simple for arithmetic storage units and ordinary-string storage units, and require little comment. However, a pictured storage unit has two interpretations, depending on the context in which it is used: it can be interpreted as a character-string storage unit or an arithmetic storage unit. Complications arise in keeping these two interpretations consistent with one another.

CHARACTER-STRING INTERPRETATION OF PICTURED STORAGE

The character-string interpretation of a pictured storage unit requires a definition of a related character-string data type. Once the related type is determined, the arithmetic operations for assignment and fetching can be defined.

Related Character Types

The related character type for a pictured storage unit is defined as:

character(i) nonvarying

where i is obtained by counting all the characters in the picture except for those that appear in the following indicators:

v the arithmetic decimal-point indicator
k the nonprinting floating-point indicator
f(n) the scale-factor indicator

These indicators are defined later; they are not counted because they contribute only to the arithmetic interpretation of a pictured storage unit.

As an example of the determination of the related character type, consider the picture attribute:

pic"s999v.99f(-12)"

This picture attribute has the related character type:

character(7) nonvarying

The length '7' is determined by counting the four characters 's999', ignoring the indicator 'v', counting the three characters '.99', and ignoring the indicator 'f(-12)'.

The following is an example of a nonnumeric picture attribute:

pic"aaxx9"

This picture attribute has the related character type:

character(5) nonvarying

The length '5' is determined by counting all the characters between the quote characters.

The data frame diagram for a pictured storage unit closely resembles that for a nonvarying character string. For the first example of a picture attribute just given, the data frame is:

```
pic " s 9 9 9 v. 9 9 f(-12) "
```

For the second example of a picture attribute the data frame is:

```
pic " a a x x 9 "
```

Observe that each data frame has a character position for each indicator that is counted in the determination of the related character type for the picture.

Character-String Assignments

The character-string assignment operation occurs when any value is assigned to a nonnumeric pictured storage unit. Before the character-string assignment operation begins, the given value is converted to the related character-string data type for the pictured storage unit. The conversion operation is not part of the assignment operation, and is covered later, in Section IV, "Value Conversion."

Because of the way the related character-string data type is determined, the converted character-string value has exactly the number of characters that are required by the picture. Each character in the converted character-string value is checked against the corresponding indicator. If a character does not satisfy the requirements of the indicator, the 'conversion' condition occurs. The 'conversion' condition is described later, in Section IV, "Value Conversion."

As an example of the assignment of a character-string value to a pictured storage unit, suppose the variable 'x' has been declared as follows:

```
del x pic "xx99999";
```

and suppose the following assignment statement is executed:

```
x = "0023.15";
```

The related character-string data type for the given picture is:

```
char(7)
```

The character string constant in the assignment statement already has exactly this data type, so no preliminary conversion is necessary. The assignment operation can begin. The picture is examined from left to right, and each indicator is checked against the corresponding character. The 'conversion' condition occurs on the fifth indicator; it is a '9', which specifies a digit and the corresponding character is a period. Therefore the assignment cannot be made.

Character-String Fetches

The character-string fetch operation occurs when the value of a numeric pictured storage unit is fetched in a context that requires a character-string value or when the value of a nonnumeric pictured storage unit is fetched in any context.

The character-string fetch operation provides a value whose data type is the related character-string data type for the picture. It can never be invalid or cause a condition to occur.

ARITHMETIC INTERPRETATION OF PICTURED STORAGE

The arithmetic interpretation of a pictured storage unit requires the definition of a related arithmetic data type. Once the related type is determined, the arithmetic operations for assignment and fetching can be defined.

Related Arithmetic Data Types

The related arithmetic data type for a given picture attribute has one of the forms:

fixed decimal(p,q)

float decimal(p)

where p and q are determined from the given picture attribute. The first form is used for a fixed-point picture and the second, for a floating-point picture.

For a fixed-point picture, the related precision attribute is determined as follows:

p is the number of digit indicators in the entire picture.

q is the number of digit indicators to the right of the 'v' indicator minus the value given by the scale-factor indicator. If there is no 'v' indicator in the picture, it is assumed to be at the right end of the picture. If there is no scale-factor indicator in the picture, it is assumed to be 'f(0)'.

As an example, consider the picture attribute:

```
pic"s999v.99f(-2)"
```

This picture has the related arithmetic data type:

```
fixed decimal(5,4)
```

The value p=5 was obtained by ignoring the 's' indicator, counting the three '9' indicators, ignoring the 'v' indicators, counting the two '9' indicators and ignoring the scale-factor indicator. The value q=4 was obtained by counting the two '9' indicators after the 'v' indicator and then subtracting the value given by the scale-factor indicator, '-2'.

For a floating-point picture, the related precision attribute is determined as follows:

p is the number of digit indicators in the mantissa of the picture; that is, before the 'e' or 'k' indicator.

As an example, consider the picture attribute:

```
pic"s9v.9999es99"
```

This picture has the related arithmetic data type:

```
float decimal(5)
```

The value p=5 was obtained by ignoring the 's' indicator, counting the '9' indicator, ignoring the 'v.' indicator, counting the four '9' indicators, and ignoring the 'e' indicator and everything after it.

Arithmetic Assignments

The arithmetic assignment operation occurs when any computational value is assigned to a numeric pictured storage unit. Before the arithmetic assignment operation begins, the given value is converted to the related arithmetic data type for the pictured storage unit. During the conversion, a condition may occur to report that the assigned value is out of range; also, the rightmost digits may be truncated or rounded off if the related arithmetic data type cannot accommodate them. The conversion operation is not part of the assignment operation, and is covered later, in Section IV, "Value Conversion."

Because of the way the related arithmetic data type is determined, the converted arithmetic value has exactly the number of digits that are required by the picture. Other parts of the pictured value are determined by the indicators in the picture, in accordance with the detailed definitions given later in this section. The result of the assignment operation is a character sequence that is placed in the pictured storage unit.

A simple example of the assignment of an arithmetic value to a pictured storage unit follows. Suppose the variable 'x' has been declared as follows:

```
dcl x pic"$99s";
```

and suppose the following assignment statement is executed:

```
x = 3;
```

The related arithmetic data type for the given picture is:

```
fixed dec(2)
```

before the assignment operation can begin, the given value must be converted to the related arithmetic data type; the result is:

```
+03.
```

The value is shown in the representation that is used to store a 'dec(2)' value. Now the assignment operation can begin. The picture is examined from left to right, and each indicator is processed as follows:

1. The first indicator is '\$'. This indicator means that a dollar sign must appear at this point in the character string. Therefore, the first character is '\$'.
2. The second indicator is '9'. This means that a decimal digit must appear, and the first digit of the converted given value is used. Therefore the second character is '0'.
3. The third indicator is '9'. This means, again, that a decimal digit must appear, and the second digit of the converted given value is used. Therefore the third character is '3'.
4. The fourth indicator is 's'. This means that the sign must appear, and the sign of the converted given value is used. Therefore the fourth character is '+'.

The resulting character sequence is:

\$03+

and this is the value that is placed in the pictured storage unit named 'x'.

Arithmetic Fetches

The arithmetic fetch operation occurs when the value of a numeric pictured storage unit is fetched in a context that requires an arithmetic or bit-string value. After the fetch is complete, some conversion of the arithmetic result may be required; but this conversion is not part of the fetch operation.

The arithmetic fetch operation is the inverse of the assignment operation. That is, if a given arithmetic value is assigned to a pictured storage unit, then a subsequent fetch operation will obtain exactly the value that was assigned.

A simple example of the fetching of an arithmetic value from a pictured storage unit follows. The example is parallel to the example just given, in the discussion of assignment. Suppose the variable 'x' has been declared as follows:

```
dcl x pic"$99s";
```

and suppose that 'x' appears in a context that requires an arithmetic value, such as the expression:

x+1

The related data type for the given picture is

fixed dec(2)

Let the current value of 'x' be the character sequence:

\$03+

In order to fetch the arithmetic value of the storage unit, the picture is examined from left to right, and each indicator is interpreted as follows:

1. The first indicator is '\$'. This indicator makes no contribution to the arithmetic interpretation and is ignored.
2. The second indicator is '9'. This indicator contributes a digit to the arithmetic interpretation. Therefore, the first digit of the arithmetic interpretation is '0'.
3. The third indicator is '9'. This indicator also contributes a digit to the arithmetic interpretation, therefore, the second digit of the arithmetic interpretation is '3'.
4. The fourth indicator is 's'. This indicator contributes the sign for the arithmetic interpretation. Therefore, the sign of the arithmetic interpretation is '+'.

The resulting arithmetic value is:

+03.

which is the correct representation for a 'fixed dec(2)' value.

Fixed-Point Pictures

There are many kinds of fixed-point pictures. However, any fixed-point picture can be constructed in three steps, as follows:

1. Start with a sequence of one or more '9' indicators. (The '9' indicator is called the no-suppression digit indicator, and means that a digit must appear in the corresponding position of the character-string value.)
2. Optionally, insert any decimal-point, sign, dollar, or insertion-character indicators. Certain restrictions on the way these indicators are used must be observed; for example, a dollar indicator must be at the beginning or end of the sequence, not between two '9' indicators.
3. Optionally, choose a zero-suppression, drifting-sign, or drifting-dollar indicator and, proceeding from left to right, replace one or more of the '9' indicators in the picture. Restrictions on the choice of the new indicator must be observed; for example, a drifting dollar indicator can be used only if the leftmost '9' is immediately preceded by a dollar indicator.

These three steps could be carried out with pencil and paper as an exercise; however, they are presented here as a convenient form of definition for the fixed-point pictures. The various restrictions mentioned in Steps 2 and 3 are given later, in the descriptions of each of the indicators.

An example of the construction of a picture according to the steps just given is:

```
9999
$99v.99
$$$v.99
```

Here, a sequence of four '9' indicators was created by Step 1; then a dollar indicator and a decimal-point indicator were inserted by Step 2; and finally a drifting-dollar digit indicator was chosen to replace some leading '9' indicators (two of them). A second example is:

```
999
999
zzz
```

Here, the optional Step 2 was skipped and the 'z' zero-suppression indicator was chosen to replace some leftmost '9' indicators (all three of them).

Once a valid fixed-point picture has been constructed, its interpretation must be determined; that is, the assignment and fetch operations must be defined. Most of the interpretation can be expressed in terms of the individual indicators that appear in the picture. However, there is one rule that pertains to the picture as a whole:

If a zero value is assigned to a fixed-point picture that does not contain a '9' indicator, then the entire character string becomes a sequence of Ø (blank) characters (if the suppression was not by a '*' indicator) or a sequence of '*' characters (if suppression was by the '*' indicator).

For example, if the picture attribute is:

```
pic"$zzv.zzs"
```

and the value is zero, then the character string value is not

```
"$ØØ.ØØ+"
```

which is the result of suppressing zeros, but rather is

```
"ØØØØØØØØ"
```

which is the result of setting all characters to blanks.

NO-SUPPRESSION DIGIT INDICATOR

The most basic of all the indicators is the no-suppression digit indicator, which is defined as follows:

- 9 means that a decimal digit must appear in the corresponding position of the character-string value. In the arithmetic interpretation, the corresponding digit is interpreted as a part of a decimal number.

The name "no-suppression digit indicator" means that this indicator specifies a digit that is never suppressed (replaced by a blank when it is zero).

An example of a pictured attribute that uses the no-suppression digit indicator is:

<u>Attribute</u>	<u>Assigned</u>	<u>Char Type and Value</u>	<u>Arith Type and Value</u>
pic"999"	180	char(3) "180"	fixed dec(3) +180.
	2	"002"	+002.
	13.99	"013"	+013.
	1000	(undefined)	(undefined)
	-22	(undefined)	(undefined)

For this set of examples, it is assumed that some variable, say 'x', has been declared with the given picture attribute:

```
dcl x pic"999";
```

For each line of the example, it is assumed that a value has been assigned to 'x'. For the first line, the assignment is:

```
x = 180;
```

Each line gives the data type and value for two contexts. The entry under "char type and value" applies when the variable is referenced in a context that requires a character-string value. The entry under "arith type and value" applies for any other context; that is, when the variable is referenced in a context that requires an arithmetic or bit-string value.

The assignment of '2' to the variable shows that leading zeros are preserved in the character string value when '9' indicators are used. The last three examples illustrate three ways in which a value can be arithmetically modified by assignment to a pictured storage unit:

- When the assigned value has more accuracy than provided for by the picture (as in the case of 13.99), the low-order digits are dropped without rounding. Although the absence of rounding is not always acceptable, this value modification is thought of as an approximation rather than an error; and no exceptional condition occurs when such an assignment is performed.
- When the assigned value is too big for the picture (as in the case of 1000), the value assigned to the storage unit is undefined. This value modification is an error and the 'size' condition occurs. Usually a 'size' condition terminates program execution, but recovery from the error is described in Section XIII, "Condition Handling."
- When the assigned value is negative and the picture does not have a sign indicator (as in the case of -22), then the value assigned to the storage unit is undefined, and the 'size' condition should occur to indicate an error. Although the program is in error, the error may not be detected by the Multics PL/I compiler and the 'size' condition may not occur.

DECIMAL-POINT INDICATOR

The decimal-point indicator is made up of two characters, and is defined as follows:

- v. means that if there is no scale-factor in the picture then the decimal point appears at this position, both in the character-string and the arithmetic value of the pictured storage unit.

The 'v.' indicator can appear no more than once and, except for intervening insertion-character indicators (described later), the indicator must be adjacent to a digit indicator. For the rare case that there is a scale factor in the picture, see the definition of the arithmetic decimal-point indicator, given later.

An example of a picture that makes use of a decimal-point indicator is:

<u>Attribute</u>	<u>Assigned</u>	<u>Char Type and Value</u>	<u>Arith Type and Value</u>
pic"999v.99"	500.98	char(6) "500.98"	fixed dec(5,2) +500.98
	16	"016.00"	+016.00
	0	"000.00"	+000.00
	6.839	"006.83"	+006.83
	1500.98	(undefined)	(undefined)
	-1	(undefined)	(undefined)

The last three assignments illustrate again the three kinds of arithmetic value modification mentioned under "The No-Suppression Digit Indicator". Replicators could have been used in the picture in any of the following ways:

pic"(3)9v.(2)9" pic"(3)9v.99" pic"999v.(2)9"

A second example of the use of the decimal-point indicator is:

<u>Attribute</u>	<u>Assigned</u>	<u>Char Type and Value</u>	<u>Arith Type and Value</u>
pic"999v."	180	char(4) "180."	fixed dec(5) +180.

Here the effect of 'v.' is to make the decimal point in the character string explicit; it could be omitted without changing the arithmetic value. Compare this example to the first line in the examples for "The No-Suppression Indicator".

Each of the two characters that make up the decimal-point indicator is, itself, an indicator. The 'v' is the arithmetic decimal-point indicator and the '.' is one of the insertion-character indicators. The separate use of 'v' and '.' is rare, but their definitions are included, for completeness, later in this section.

SIGN INDICATORS

A picture can contain a sign indicator. The sign indicators are defined as follows:

- s means that a '+' or '-' must appear in the corresponding character position. In the arithmetic interpretation, the characters '+' and '-' mean the value is positive or negative, respectively.
- + means that a '+' or a blank must appear in the corresponding character position. In the arithmetic interpretation, the characters '+' or blank mean the value is positive or negative, respectively.
- means that a blank or a '-' must appear in the corresponding character position. In the arithmetic interpretation, the characters blank or '-' mean the value is positive or negative, respectively.
- cr means that two blanks or 'cr' must appear in the corresponding two character positions. For the arithmetic interpretation, the two blanks or 'cr' mean the value is positive or negative, respectively.
- db means that two blanks or 'db' must appear in the corresponding two character positions. For the arithmetic interpretation, the two blanks or 'db' means the value is positive or negative, respectively. Despite the opposite mnemonic significance of 'cr' (credit) and 'db' (debit), both are used to indicate negative arithmetic values..

Only one sign indicator can occur in a fixed-point picture. Any one of the indicators

s + -

can occur before the first digit indicator in the picture; this leading position is in accord with the common usage of signs. Any one of the indicators

s + - cr db

can occur after the last digit indicator; this is in accord with some business usage.

Examples of the use of a sign indicator at the beginning of a picture are:

<u>Attribute</u>	<u>Assigned</u>	<u>Char Type and Value</u>	<u>Arith Type and Value</u>
pic"s999"	82	char(4) "+082"	fixed dec(3) +082.
	0	" +000"	+000.
	-82	" -082"	-082.
pic"+999"	82	char(4) "+082"	+082.
	0	" +000"	+000.
	-82	"Ø082"	-082.
pic"-999"	82	char(4) "Ø082"	fixed dec(3) +082.
	0	"Ø000"	+000.
	-82	" -082"	-082.

Two of these assignments are of particular interest. The assignment of a negative value to a variable with a '+' sign indicator gives a blank for the sign (second picture, third assignment); this is not a common handling of the sign, and the use of the '+' indicator is not recommended. In contrast, the assignment of a number to a variable with the '-' indicator (third picture) follows the familiar conventions for the sign, and the use of the '-' is a useful alternative to the 's' indicator.

Examples of the use of a sign indicator at the end of a picture are:

<u>Attribute</u>	<u>Assigned</u>	<u>Char Type and Value</u>	<u>Arith Type and Value</u>
pic"99v.99s"	5 -5	char(6) "05.00+" "05.00-"	fixed dec(4,2) +05.00 -05.00
pic"99v.99+"	5 -5	char(6) "05.00+" "05.00%"	fixed dec(4,2) +05.00 -05.00
pic"99v.99-"	5 -5	char(6) "05.00%" "05.00-"	fixed dec(4,2) +05.00 -05.00
pic"99v.99cr"	5 -5	char(7) "05.00%" "05.00cr"	fixed dec(4,2) +05.00 -05.00
pic"99v.99db"	5 -5	char(7) "05.00%" "05.00db"	fixed dec(4,2) +05.00 -05.00

As before, the '+' indicator behaves contrary to familiar conventions. The 'cr' (credit) and 'db' (debit) indicators are designed especially for business programming and behave in accordance with accounting conventions.

DOLLAR INDICATOR

A picture can contain a dollar indicator, which is defined as follows:

\$ means that a '\$' must appear in the corresponding position of the character string value. In the arithmetic interpretation, the indicator is ignored.

A dollar indicator can occur in either of two ways in a fixed-point picture. First, it can appear anywhere before the first digit position. Second, it can occur anywhere after the last digit position and before a 'cr', 'db', 's', '+', or '-' indicator (if there is one).

Examples of the use of the dollar indicator are:

<u>Attribute</u>	<u>Assigned</u>	<u>Char Type and Value</u>	<u>Arith Type and Value</u>
pic"\$999v.99"	20	char(7) "\$020.00"	fixed dec(5,2) +020.00
pic"999v.\$db"	-478	char(7) "478.\$db"	fixed dec(3) -478.

ZERO-SUPPRESSION DIGIT INDICATORS

A fixed-point picture can contain zero-suppression digit indicators, which are defined as follows:

- z means that a decimal digit or a blank occurs in the corresponding position of the string value. The blank occurs if the character would be a leading zero; otherwise, a decimal digit occurs. In the arithmetic interpretation, a blank that corresponds to the indicator is interpreted as a zero.
- * means that a decimal digit or a '*' occurs in the corresponding position. The definition is parallel to that for 'z'.
- y means that a decimal digit or a blank occurs in the corresponding position of the string value. The blank occurs if the corresponding character would be zero (regardless of whether or not it would be a leading zero). In the arithmetic interpretation, a blank that corresponds to the indicator is interpreted as a zero.

A leading zero is a digit that is a zero, is not preceded by a nonzero digit, and (less obviously) is to the left of a decimal-point indicator, 'v.' or 'v', if a decimal-point indicator appears.

Each of these zero-suppression digit indicators can be used as an alternative to a no-suppression digit indicator, '9', in a fixed-point picture. The first two indicators are subject to the following restrictions:

- The 'z' indicators in a picture must not be preceded by any other kind of digit indicator; similarly, the '*' indicators must not be preceded by any other kind of digit indicator.
- If a 'z' indicator is used to the right of a 'v.' or 'v' indicator, then all digit indicators in the fixed-point picture must be 'z' indicators. Similarly, if a '*' is used to the right of a 'v.' or 'v', then all digit indicators must be '*' indicators.

These restrictions can be illustrated by considering the following picture:

```
pic"s99v.99"
```

The only valid pictures that have the same pattern of digit indicators but use 'z' or '*' are:

```
pic"sz9v.99"  
pic"szzv.99"  
pic"szzv.zz"
```

```
pic"s*9v.99"  
pic"s**v.99"  
pic"s**v.**"
```

Examples of the suppression of leading zeros by means of the 'z' indicator are:

<u>Attribute</u>	<u>Assigned</u>	<u>Char Type and Value</u>	<u>Arith Type and Value</u>
pic"-zzv.zz"	-12.18	char(6) "-12.18"	fixed dec(4,2) -12.18
	-2.18	"-Ø2.18"	-02.18
	-0.18	"-ØØ.18"	-00.18
	-0.08	"-ØØ.08"	-00.08
	-0.00	"ØØØØØØ"	+00.00
pic"-z9v.99"	-0.18	"-Ø0.18"	-00.18
	-0.00	"+Ø0.00"	+00.00

Observe that, in the last assignment for the first picture, all of the digits are suppressed and therefore the entire string value is blank.

Examples of the suppression of leading zeros by means of the '*' indicator are:

<u>Attribute</u>	<u>Assigned</u>	<u>Char Type and Value</u>	<u>Arith Type and Value</u>
pic"\$**v.**"	56.20	char(6) "\$56.20"	fixed dec(4,2) +56.20
	0.03	"\$**0.03"	+00.03
	0	"*****"	+00.00

When the value is not zero, as in the second example, the zeros following the decimal indicator are not suppressed. When the value is zero, as in the third example, all the zeros are suppressed and the entire string is filled with asterisks.

Examples of the suppression of zeros by means of the 'y' indicator are:

<u>Attribute</u>	<u>Assigned</u>	<u>Char Type and Value</u>	<u>Arith Type and Value</u>
pic"99y99"	44444	char(5) "44444"	fixed dec(5) +44444.
	44044	"44Ø44"	+44044.
	5	"00Ø05"	+00005.
pic"yyy99"	10000	char(5) "1ØØ00"	+10000.

DRIFTING-SIGN DIGIT INDICATORS

A fixed-point picture can contain drifting-sign digit indicators, which are defined as follows:

- s has the same meaning as 'z', except that it indicates that the sign ('+' or '-') must be moved to the right over a suppressed leading zero.
- +
-

When a sequence of 's' indicators appears in a picture, the leftmost one is interpreted as a sign indicator and the remaining ones are interpreted as drifting sign digit indicators. The same interpretation is applied to a sequence of '+' or '-' indicators. Drifting-sign digit indicators are subject to the same restrictions as the 'z' zero-suppression indicator.

Some examples of the use of drifting-sign digit indicators are:

<u>Attribute</u>	<u>Assigned</u>	<u>Char Type and Value</u>	<u>Arith Type and Value</u>
pic"sss9v.99"	-180.39	char(7) "-180.39"	fixed dec(5,2) -180.39
	-.39	"BB-0.39"	-000.39
pic"----v.--"	-180.39	char(7) "-180.39"	fixed dec(5,2) -180.39
	-.39	"BBB-.39"	-000.39
	-.09	"BBB-.09"	-000.09
	-0	"BBBBBBB"	+000.00

The two attributes just given are similar to the following:

pic"szz9v.99"

pic"-zzzv.zz"

The only difference is in the placement of the character that represents the sign. When drifting-sign digit indicators are used, the sign character occupies the position of the rightmost suppressed leading zero. When the 'z' indicator is used, the sign character remains at the position specified by the sign indicator and (in these examples) is the first character.

DRIFTING-DOLLAR DIGIT INDICATOR

A fixed-point picture can contain drifting-dollar digit indicators, defined as follows:

\$ has the same meaning as 'z', except that it indicates that the dollar character must be moved to the right over a suppressed leading zero.

When a sequence of '\$' indicators appears in a picture, the leftmost one is interpreted as a dollar indicator and the remaining ones are interpreted as drifting-dollar digit indicators. Drifting-dollar digit indicators are subject to the same restrictions as the 'z' zero suppression indicators.

Some examples of the use of drifting-dollar digit indicators are:

<u>Attribute</u>	<u>Assigned</u>	<u>Char Type and Value</u>	<u>Arith Type and Value</u>
pic"\$\$\$9v.99"	200	char(7) "\$200.00"	fixed dec(5,2) +200.00
	.03	"BB\$0.03"	+000.03
	0	"BB\$0.00"	+000.00
pic"\$\$\$v. \$\$"	200	char(7) "\$200.00"	fixed dec(5,2) +200.00
	.03	"BBB\$.03"	+000.03
	0	"BBBBBBB"	+000.00

INSERTION-CHARACTER INDICATORS

A picture can contain insertion-character indicators, which are defined as follows:

- . means that a period must appear in the corresponding character position unless zero suppression applies.
- , means that a comma must appear in the corresponding character position unless zero suppression applies.
- / means that a slash must appear in the corresponding character position unless zero suppression applies.
- b means that a blank must appear in the corresponding character position unless zero suppression applies. Observe that the indicator is the letter 'b' but it inserts a blank character.

In the arithmetic interpretation of a pictured storage unit, all insertion characters are ignored. There is no restriction on the way insertion characters are placed in a picture.

The definitions just given refer to the possibility that zero suppression applies to an insertion character, and that possibility is now described. An insertion character is suppressed when it immediately follows a suppressed digit or another suppressed insertion character; however, no suppression occurs to the right of a 'v' indicator. The effect of suppression is the same as for a digit; that is, the insertion character is replaced by a '*' or a blank, depending on whether the indicator that caused the neighboring zero suppression was a '*' or some other indicator.

Some examples of pictures that contain insertion-character indicators follow:

<u>Attribute</u>	<u>Assigned</u>	<u>Char Type and Value</u>	<u>Arith Type and Value</u>
pic"\$9,999v.99"	1529.09	char(9) "\$1,529.09"	dec(6,2) +1529.09
	529.09	"\$0,529.09"	+0529.09
	.09	"\$0,000.09"	+0000.09
	0	"\$0,000.00"	+0000.00
pic"\$z,zzzv.zz"	1529.09	"\$1,529.09"	+1529.09
	529.09	"\$zz529.09"	+0529.09
	.09	"\$zzzz.09"	+0000.09
	0	"\$zzzzzzzz"	+0000.00
pic"\$\$,\$\$\$v.\$\$"	1529.09	"\$1,529.09"	+1529.09
	529.09	"\$\$529.09"	+0529.09
	.09	"\$\$\$\$\$.09"	+0000.09
	0	"\$\$\$\$zzzz"	+0000.00

Each picture uses two insertion-character indicators: one comma and one period. The examples have the following features:

- Since the only digit indicators in the first picture are '9' indicators, no zero suppression occurs. Since no zero suppression occurs, the insertion characters are never suppressed.
- In the second and third pictures, zero suppression occurs. When the first digit is zero, it is suppressed and, as a result, the comma that follows is also suppressed.
- The period insertion character is never suppressed because it appears to the right of the 'v' indicator. The 'v' indicator stops zero suppression and therefore also stops suppression of insertion characters.
- When zero is assigned to the second or third picture, the entire character string is suppressed; this occurs because all of the digit indicators are zero-suppression indicators.

In this example, the period is discussed as an insertion character; earlier, in the discussion of the decimal-point indicator 'v.', the period was considered to be a part of a two-character indicator. These two views are consistent with one another, as the discussion of the "Arithmetic Decimal-Point Indicator", given later in this section, will show.

Since there is no restriction on the way insertion-character indicators are placed in a picture, a considerable variety of effects can be achieved. Examples are:

<u>Attribute</u>	<u>Assigned</u>	<u>Char Type and Value</u>	<u>Arith Type and Value</u>
pic"99/99/99"	12075	char(8) "01/20/75"	dec(6) +012075.
pic"99.99.99"	12075	char(8) "01.20.75"	dec(6) +012075.
pic"9,999,999"	1234567	char(9) "1,234,567"	dec(7) +1234567.
pic"9b999b999"	1234567	char(9) "1 b 234 b 567"	dec(7) +1234567.

The purpose of the insertion characters is to permit various special notations for values. Insertion characters should be used conservatively, and tricks should be avoided.

ARITHMETIC DECIMAL-POINT INDICATOR

A fixed-point picture can contain an arithmetic decimal-point indicator, which is defined as follows:

- v does not have a corresponding character position and thus does not contribute a character to the character string value. In the arithmetic interpretation, the indicator determines the position of the decimal point.

The 'v' indicator can occur no more than once in a picture and, except for intervening insertion-character indicators, it must be adjacent to a digit indicator. Usually the 'v' indicator is used in conjunction with the '.' insertion-character indicator and the pair is thought of as a single indicator as described earlier under "The Decimal-Point Indicator".

The 'v' indicator is used separately in order to cause the automatic scaling of values that are assigned to a pictured storage unit. Examples are:

<u>Attribute</u>	<u>Assigned</u>	<u>Char Type and Value</u>	<u>Arith Type and Value</u>
pic"999v99"	180.98	char(5) "18098"	fixed dec(5,2) +180.98
pic"999.99"	180.98	char(6) "001.80"	fixed dec(5,0) +00180.
pic"9.99v99"	180.98	char(6) "1.8098"	fixed dec(5,2) +180.98

In each of these examples, the character string value represents a different number than the arithmetic value.

SCALE-FACTOR INDICATOR

A fixed-point picture can end with a scale-factor indicator, which is defined as follows:

$f(\underline{n})$ does not have the corresponding character positions and, therefore, does not affect the related character type of a pictured storage unit. In the arithmetic interpretation, it adds \underline{n} to the scale factor implied by the picture. The arithmetic value of a picture is $10^{*\underline{n}}$ times the apparent character string value of the picture.

Some examples of the use of the scale-factor indicator in a fixed-point picture are:

<u>Attribute</u>	<u>Assigned</u>	<u>Char Type and Value</u>	<u>Arith Type and Value</u>
pic"99v.999f(0)"	23	char(6) "23.000"	fixed dec(5,3) +23.000
pic"99v.999f(2)"	700	char(6) "07.000"	fixed dec(5,1) +0700.0
pic"99v.999f(-1)"	.04	char(6) "00.400"	fixed dec(5,4) +0.0400

An interesting feature of these examples is the fact that, for each of the three examples, the same sequence of digits appear in the character-string value and the arithmetic value. Once the related arithmetic data type has been correctly determined, it places the decimal point in the arithmetic value.

Floating-Point Pictures

A floating-point picture is composed of a sequence of four parts, as follows:

1. The mantissa, which can be any fixed-point picture that does not have a sign indicator after the digit indicators and does not have a dollar indicator.
2. A floating-point indicator, with 'e' or 'k', as defined later.
3. The exponent, which is a severely restricted fixed-point picture that begins with an optional sign indicator and has from one to three 'z' or '9' digit indicators.
4. An optional scale-factor indicator.

The floating-point indicator distinguishes a floating-point picture; that is, any picture that contains an 'e' or 'k' is necessarily a floating-point picture.

Most of the interpretation of a floating-point picture depends on the interpretation of the fixed-point pictures that are used as its mantissa and exponent. However, there are a few special rules:

- Unless the arithmetic value of the assigned value is zero, the exponent is chosen so that the first digit of the mantissa is not zero.
- Before an arithmetic value is assigned to a pictured storage unit that does not have enough digits for an exact representation, the rightmost digits are dropped by rounding. This contrasts with the truncation without rounding that is used for a fixed-point picture, described earlier under "The No-Suppression Indicator".

Examples of the application of these rules are:

<u>Attribute</u>	<u>Assigned</u>	<u>Char Type and Value</u>	<u>Arith Type and Value</u>
pic"-9v.999es99"		char(10)	float dec(4)
	-100	"-1.000e+02"	-0001.e+2
	-.01	"-1.000e-02"	-0001.e-2
	5.0025	"+5.003e+00"	+5003.e-3

FLOATING-POINT INDICATORS

A floating-point picture must contain one of the following indicators:

- e means that an 'e' must appear in the corresponding position of the character-string value. In the arithmetic interpretation, it separates the mantissa and the exponent.
- k does not have a corresponding character position and thus does not contribute a character to the character-string value. In the arithmetic interpretation, the indicator is equivalent to 'e'.

The following examples show the difference between the two kinds of floating-point indicators:

<u>Attribute</u>	<u>Assigned</u>	<u>Char Type and Value</u>	<u>Arith Type and Value</u>
pic"s9v.9999es99"	15	char(11) "+1.5000e+01"	float dec(5) +00015.e0
pic"s9v.9999ks99"	15	char(10) "+1.5000+01"	float dec(5) +00015.e0

SCALE-FACTOR INDICATOR

A floating-point picture can end with a scale-factor indicator, which is defined as follows:

f(n) does not have corresponding character positions and therefore does not affect the related character type of a pictured storage unit. The arithmetic value of a picture is 10^{**n} times the apparent value of the character string value of the picture.

Some examples of the use of the scale-factor indicator in a floating-point picture are:

<u>Attribute</u>	<u>Assigned</u>	<u>Char Type and Value</u>	<u>Arith Type and Value</u>
pic"s9v.99es99"	15	char(9) "+1.50e+01"	float dec(3) +015.e0
pic"s9v.99es99f(4)"	15	char(9) "+1.50e-03"	float dec(3) +015.e0
pic"s9v.99es99f(-3)"	15	char(9) "+1.50e+04"	float dec(3) +015.e0

Consider the second example. The arithmetic value '15' is assigned to the storage unit. It must be multiplied by 10^{**n} to get the character string representation; then when it is fetched as an arithmetic value and multiplied by 10^{**n} its value is the same as it was originally.

Complex Pictures

Pictured storage for a complex value is declared by combining any numeric picture attribute with the 'complex' attribute. The effect of the 'complex' attribute is to make the character-string value twice as long as the numeric picture requires, and to thus provide for the real and imaginary parts. The resulting character string is different from other PL/I representations of a complex value because it does not have an 'i' at the end of the imaginary part.

An example of the declaration of a pictured complex storage unit is:

<u>Attributes</u>	<u>Assigned</u>	<u>Char Type and Value</u>	
pic"s9v.999es99" complex	3-2i	char(20)	
		"+3.000e+00-2.000e+00"	
		<u>Arith Type and Value</u>	
		complex float dec(4)	
		+0003.e0-0002.e0i	

Non-numeric Pictures

The design of a nonnumeric picture is much simpler than that of a numeric picture. The nonnumeric pictures are present so that a programmer can describe a storage unit that is designed to hold a character string and to have only a character-string interpretation. A nonnumeric picture is an alternative to the use of the 'character' attribute.

NON-NUMERIC INDICATORS

A nonnumeric picture is made up of any number of nonnumeric indicators in any order; however, at least one 'x' or 'a' indicator must appear. The indicators are defined as follows:

- x means any ASCII character can appear in the corresponding character position.
- a means that a letter (upper or lower case) or a blank must appear in the corresponding character position.
- 9 means that a digit or a blank must appear in the corresponding character position.

Observe that in a numeric picture a '9' specifies a digit only, while here, in a nonnumeric picture, '9' specifies a digit or a blank.

Some examples of nonnumeric pictures are:

<u>Attribute</u>	<u>Assigned</u>	<u>Char Type and Value</u>	<u>Arith Type and Value</u>
pic"xxxxx"	"ab--3"	"ab--3"	(not applicable)
pic"aaxx9"	"ab--3"	"ab--3"	(not applicable)
	" xxxxxx "	" xxxxxx "	
	"7b--3"	(conversion)	

The last line shows that an attempt to assign a digit to a position controlled by an 'a' indicator causes the 'conversion' condition to occur. See Section XIII, "Condition Handling," later in this manual.

Guidelines for Pictured Storage

Pictured storage is ideal for programming applications in which the format and layout of input/output is important and calculations are relatively simple. Business programs often fit this description. Simple programs for teaching or one-shot execution also can make good use of pictured storage.

When a pictured variable is involved in complicated and repeated arithmetic calculations, some significant extra costs can accrue. The fetch of the arithmetic value of a pictured variable can be one or two orders of magnitude more costly than the fetch of the value of a similar arithmetic variable. In such a situation, there are four options:

- Accept the extra cost and keep the pictured variable. This is the approach often adopted in business programming, where the cost of input/output overshadows any computational costs.
- Keep the pictured variable, but introduce an arithmetic variable for use in calculation. In this case, the pictured variable is used for input/output, the arithmetic variable is used in calculations, and the values are assigned back and forth between the two variables when necessary. This approach is especially useful when input/output is performed in terms of records, as described later in Section XV, "Record Input/Output," rather than in terms of a stream.
- Drop the pictured variable, but use the picture format item in conjunction with edit-directed input/output, as described later in Section XIV, "Stream Input/Output." This approach is similar to the previous one, except that the pictured variable is a system temporary that is controlled by the PL/I processor rather than the program.
- Drop the use of the picture entirely. In this case, input is controlled by some other mechanism of input/output, usually within the extensive facilities for stream input/output. This approach is often used in short scientific programs.

In a large program, all four of these approaches might appear as a result of the separate consideration of each variable.

ADDRESS STORAGE

PL/I permits an address to be treated as a data value. Although these values are not subject to calculation in the usual sense, they can be stored, fetched, compared with one another and, ultimately, used as addresses.

Address values are discussed earlier, in Section II, "Values." There are six types of address values, as follows:

label	The address of a statement that can be the destination of a 'goto' statement
entry	The address of a statement that is an entry to a procedure
format	The address of a 'format' statement
pointer	The address of a storage unit, expressed as a full Multics address
offset	The address of a storage unit, expressed relative to the beginning of an area
file	The address of a file-state block, which, in turn, gives access to a Multics file.

For each of these types of address value, there is a corresponding type of storage unit. These storage units are described here.

A label, entry, or format address is a special kind of program address. It designates not only a certain statement of a program but also a particular activation of the block in which that statement appears. This feature is significant only when a program is recursive; it is discussed later, in Section XII, "Procedure Invocation."

Address Attributes

The data type of an address storage unit is specified by one of the following keywords:

```
label
entry
format
pointer
offset
file
```

Sometimes other keywords are used in close association with the data type attribute in the declaration of an address variable: however, those other keywords are not a part of the storage type. For example, consider:

```
dcl L1 label local;
dcl L2 label;
```

Here, the storage type of both 'L1' and 'L2' is just 'label'. The keyword 'local' is a usage attribute, and is not part of the storage type; its effect is described in Section XI, "Program Flow." As a second example, consider:

```
dcl P1 entry(float);
dcl P2 entry(char(20), dim(3) fixed) returns(char(20));
```

Here the storage type of both 'P1' and 'P2' is 'entry'. The remainder of the declarations is, again, information about the usage of the address variables.

One of the address data attributes can be abbreviated, as follows:

<u>keyword</u>	<u>abbreviation</u>
pointer	ptr

Example of Address Storage

As an example, consider the variable 'start' whose declaration is:

```
dcl start ptr;
```

The storage type of 'start' is

pointer

and its storage can be diagrammed as follows:

start 

The data frame is shown as a single box, without any division into parts. A Multics pointer is divided into a segment number, a word offset, and so on; but these details of structure are irrelevant from the point of view of PL/I.

AREA STORAGE

Area storage is used in connection with the PL/I facilities for storage management. In contrast to all other PL/I variables, a variable storage unit of type 'area' is not used directly to store data; instead, it is a reservoir of storage that supplies storage for the allocation of other variables. The use of area storage is an advanced and specialized feature of PL/I, and many applications do not require area storage.

When an 'area' variable is defined, the program specifies only the number of words occupied by the variable. In its initial form, the 'area' variable is empty. As execution of the program proceeds, variables are allocated within the area, are used, and are eventually freed. At some time after its last use, the 'area' variable itself is freed. The allocation and freeing of variables is discussed in detail later, in Section VII, "Storage Management."

When a variable is allocated within an 'area' variable, some words of the 'area' variable are occupied by the allocated variable; then, when the variable is freed, the words become available again. The PL/I processor automatically keeps data that show which words are occupied at any time; this data is kept in the 'area' variable itself, and thus uses up a portion of the storage occupied by the 'area' variable. Thus an 'area' variable is more than just a block of storage: it is a complete storage system that is embedded in a larger system.

An important feature of an 'area' variable is that it defines its own system of addresses; that is, the address of a variable that is allocated within an area can be expressed as an offset relative to the first word of the area. When the contents of one 'area' variable is copied into another 'area' variable, the Multics addresses of the variables contained in the area change, but the offset addresses relative to the area do not change. This feature of 'area' variables is important wherever offset variables are used in forming linked lists of data objects.

'area' Attribute

The data type of an area storage unit has the following form:

```
area( as )
```

where as is the area size. The area size must be an extent; that is, it must have the form:

```
exp  
exp refer (ref)  
*
```

where exp is an expression and ref is a reference. The first form is used in most cases; it must yield a value that can be converted to a 'fixed binary(19)' value. The second two forms are described later, in Sections VII and XII, "Storage Management" and "Procedure Invocation," respectively.

The value of the area size at the time the area is allocated determines the number of words that are allocated for the area. The size of an area cannot be greater than the size of a Multics segment, $2^{*}18$ words. In general, an area is not allocated at the beginning of a segment and it must be small enough to fit in the portion of the segment that remains. An 'area' variable is allocated just like any other variable; details are given later, in Section VII, "Storage Management."

The capacity of an 'area' variable is always somewhat less than the area size; this is because a portion of the storage allocated for the area variable is used as the occupation record; that is, the data that show which words of the area are in use and which are free. For example, consider the declaration:

```
dcl A1 area(1000);
```

According to this declaration, the 'area' variable 'A1' occupies exactly 1000 words of Multics storage. Suppose that many variables with the storage type

```
char(40) nonvarying
```

must be allocated in 'A1'. This character variable occupies 10 words of storage; however, it is not possible to allocate 100 such variables in 'A1' because of the storage used by the occupation record.

Default Rule

The following default rule applies to the 'area' attribute:

<u>Omitted item</u>	<u>Default</u>
area size	1024

The use of this default is not recommended. The area size is an important parameter, and should be carefully selected and explicitly given.

Example of Area Storage

As an example, consider the following declaration:

```
dcl M area(2*n);
```

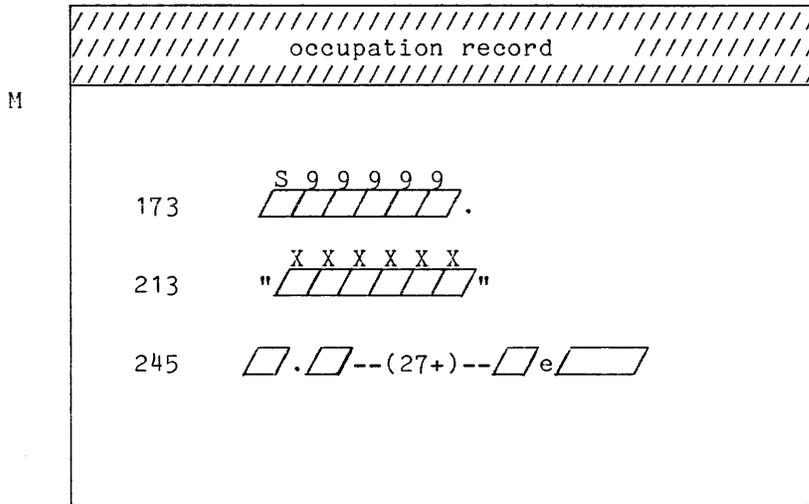
Suppose n=4096 when 'area' is allocated. Then the storage type of 'M' is:

```
area(8192)
```

The area variable occupies 8192 Multics words. As program execution proceeds, variables can be allocated, used, and freed within 'M'. Suppose that, at a particular time in program execution, three variables with the following storage types are allocated to 'M':

```
fixed dec(5)  
char(6)  
float
```

At this time, 'M' can be diagrammed as follows:



The number given to the left of each storage unit in the area is the word offset of the storage unit relative to the beginning of the area. The three variables shown in the diagram do not, of course, exhaust the capacity of the area, which is 8192 words minus the words used for the occupation record.

AGGREGATE STORAGE

It is often useful to gather together a set of scalar variables, arrange them in a sequence, and treat them as a single variable. Such a variable is an aggregate. There are two kinds of aggregate, the structure and the array, and these types of variables are described here.

Once aggregates have been introduced, it is necessary to distinguish between a variable that is contained in another, larger variable and one that is not. A variable that is contained in an aggregate variable is a minor variable, and is said to be a component of the aggregate. A variable that is not contained in another variable is a major variable. Sometimes a major variable is called a level-one variable; that terminology arose from the way structures are declared. Many examples of major and minor variables are given in the discussion that follows.

Structures

A structure variable is a sequence of members. The structure itself has a name and, in addition, each of the members of the structure has its own name. When a program operates on the entire structure, the structure name alone is used as the reference. When a program operates on a member of the structure, both the structure name and the member name, separated by a period, are used as the reference. (Often the member name alone can be used, but that is just an abbreviation of the complete reference.)

Each member of a structure can have any storage type. For example, a structure could be declared that has an arithmetic scalar as its first member, a structure of string variables as its second member, and an array of arithmetic variables as its third member. An example of such a structure is given in the following discussion of "Level Numbers".

LEVEL NUMBERS

The structure is the only part of the storage type that is not given by attributes; instead, it is given in quite a different way, by level numbers. The level number is written just before each name used in the declaration of a structure, whereas the attributes are written after the name. The level number of each member of a structure variable must be greater than the level number of the structure itself; that is how the hierarchy is indicated. It is recommended that a major variable have level number one, a member of a major variable have level number two, a member of a member should have level number three, and so on, to whatever depth is required.

For purpose of discussion, consider the following declaration of a structure:

```
dcl 01 S1,  
    02 alpha dec(5),  
    02 beta,  
        03 x char(4),  
        03 y char(6),  
    02 gamma(3) float;
```

With the help of the level numbers, '01', '02', and '03', the PL/I processor can recognize that this statement is the declaration of nine variables, as follows. The first variable is designated by:

S1 (designates the major structure variable)

The designators of the three members of 'S1' are:

S1.alpha (designates a minor scalar variable)
S1.beta (designates a minor structure variable)
S1.gamma (designates a minor array variable)

The designators of the two members of 'S1.beta' are:

S1.beta.x (designates a minor scalar variable)
S1.beta.y (designates a minor scalar variable)

The designators of the three elements of 'S1.gamma' are:

S1.gamma(1) (designates a minor scalar variable)
S1.gamma(2) (designates a minor scalar variable)
S1.gamma(3) (designates a minor scalar variable)

This example shows how a member of a structure can be a scalar, a structure, or an array. (The use of an array here anticipates the description of arrays that appears later in this section; however, the array 'S1.gamma' is a simple one, and its treatment should be obvious.)

Two points of programming style arise in connection with the declaration of structures.

- First, although the PL/I processor relies upon level numbers to determine the structure, a programmer relies upon the layout of the declaration. Therefore, the indentation shown in the example 'declare' statement should be used.
- Second, although the PL/I processor does not require a leading zero on a level number, the use of such a zero adds emphasis and distinguishes the level number from a computational constant.

STRUCTURE STORAGE TYPES

The storage type of a structure is obtained from the declaration in three steps:

1. Omit the name of the structure and the names of its members.
2. Normalize the level numbers.
3. Obtain the storage type of each member, depending on whether the member is a scalar, a structure, or an array.

The levels of a structure are normalized by reducing them until the structure as a whole has level number one, the members of the structure each have level number two, and so on.

The members of a structure are arranged in storage in the order in which they appear in the declaration. Thus the order of the members of the structure 'S1' is:

```
S1.alpha
S1.beta
S1.gamma
```

Similarly, the order of the members of S1.beta is:

```
S1.beta.x
s1.beta.y
```

As an example of the determination of the storage type of a structure, consider 'S1.beta' as declared in the preceding description of level numbers. The declaration was given as:

```
02 beta,
   03 x char(4),
   03 y char(6)
```

Omission of the variable names gives:

```
02,
   03 char(4),
   03 char(6)
```

Normalization of the level numbers gives:

```
01,
   02 char(4),
   02 char(6)
```

This is the desired result: the storage type for 'S1.beta'.

EXAMPLES

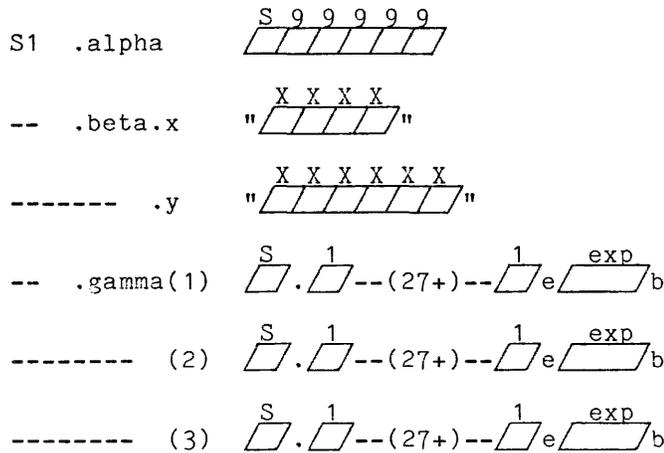
As a detailed example of storage for a structure, consider once again the variable 'S1', which has been used throughout the discussion of structures. Its declaration is:

```
dcl 01 S1,
   02 alpha fixed dec(5),
   02 beta,
       03 x char(4),
       03 y char(6),
   02 gamma(3) float bin;
```

This statement declares nine variables, and their storage types are as follows:

<u>Designator</u>	<u>Storage Type</u>
S1	01, 02 fixed dec(5), 02, 03 char(4), 03 char(6), 02 float bin
S1.alpha	fixed dec(5)
S1.beta	01, 02 char(4), 02 char(6)
S1.beta.x	char(4)
S1.beta.y	char(6)
S1.gamma	dim(3) float bin
S1.gamma(1)	float bin
S1.gamma(2)	float bin
S1.gamma(3)	float bin

The storage for 'S1' can be diagrammed as follows:



Thus 'S1' is made up of six scalar storage units. The arrangement of these storage units on six separate lines does not imply that they occupy six words of a Multics segment. The mapping of storage units into segment words is done according to rules that are given later in this section, when "Alignment" is discussed. Because unused bits or bytes of storage are sometimes placed between the storage required for the storage units, the rules are not simple.

As a second example, consider the variable 'account', declared as follows:

```
    decl 01 account,
          02 name,
            03 last char(30) var,
            03 middle char(1),
            03 first char(20) var,
            03 title char(4) var,
          02 number pic"9999",
          02 address,
            03 street char(30),
            03 loc,
              04 city char(20),
              04 state char(2),
              04 zip pic"99999",
          02 balance pic"$,,$$,,$$9v.99",
          02 credit_limit pic"$,,$$,,$$9v.99";
```

This example shows how a simple financial record for a member of an organization can be handled. The following observations apply:

- The entire structure of 11 scalar components can be referred to by a single name, 'account'. This reference is convenient when, for example, the structure is written as a record in a permanent file.
- The entire name of the individual who has the account can be referred to by a single designator, 'account.name'. This reference is convenient when, for example, a list of account holders must be prepared.
- The title (Mr., Mrs., etc.) and the last name can be accessed individually by 'account.name.title' and 'account.name.last'. This reference is convenient when, for example, the greeting of a letter must be prepared.

These observations are intended to emphasize the fact that the components of a structure can be handled collectively or separately, according to the requirements of each operation. A complete description of references to structures is given later, in Section VIII, "Expressions."

Arrays

An array variable is a sequence of elements. The variable has a single name, and the individual elements are designated by giving the array name and a list of one or more subscripts. When an array variable is referenced in a program, general expressions can be given for the subscripts, and the specific subscript values are then determined each time the reference is evaluated. In this way, data addresses can be calculated during program execution, and a single array reference can designate different elements at different times.

All elements of a given array have the same storage type. That storage type can specify a scalar or structure variable, but not an array variable. In other words, the storage type of an element of an array can specify any variable except an array.

An array variable is declared by means of a dimension attribute. The dimension attribute should be inserted just after the name of the variable. Consider the following declarations:

```
dcl m1 float bin;

dcl 01 m2,
    02 a fixed bin,
    02 b char(32);
```

These are declarations of a scalar variable and a structure variable, respectively. The first can be changed into a declaration of an array of scalars by the addition of an appropriate 'dimension' attribute, and the second can be changed into an array of structures in the same way; thus:

```
dcl m1 dimension(1:20) float bin;

dcl 01 m2 dimension(1:20),
    02 a fixed bin,
    02 b char(32);
```

The discussion of the 'dimension' attribute that follows gives the interpretation of these declarations and also gives the abbreviations and defaults that permit them to be written in shorter forms, namely:

```
dcl m1(20) float bin;

dcl 01 m2(20),
    02 a fixed bin,
    02 b char(32);
```

'dimension' ATTRIBUTE

The 'dimension' attribute has the following form:

```
dimension( bplist )
```

where bplist is the bound-pair list, and is a sequence of one or more bound-pairs separated by commas. The number of bound-pairs is the dimensionality of the attribute; it determines how many subscript positions are associated with the array. For example, consider:

```
dcl A dimension(1:8,j-1:2*(n+1)) float;
```

According to this declaration, 'A' has a dimensionality of two, and its bound-pairs are '1:8' and 'j-1:2*(n+1)'.

The purpose of a bound-pair is to specify the range for a given subscript position. Each bound-pair has the one of the forms:

```
lb : hb
```

*

where lb and hb are the lower and higher bounds. The second form is described later in Section XII, "Procedure Invocation." The bound-pair specifies that the subscript has the following sequence of values:

```
lb, lb+1, lb+2, ..., hb
```

Consider, again, the following example:

```
dcl A dimension(1:8,j-1:2*(n+1)) float;
```

The 'dimension' attribute specifies the following range of integers for the first subscript position:

1, 2, 3, ..., 8

The second bound-pair of 'A' depends on variables, and such variables are evaluated when the array variable is allocated or, if the variables are based, whenever the array variable is referenced. Suppose the variables are $j=4$ and $n=2$ at the time of allocation. Then the attribute specifies the following range for the second subscript position:

3, 4, 5, 6

It follows that the array named 'A' is made up of $8*4 = 32$ elements.

Each bound in a dimension attribute must be an extent, which has one of the following forms:

exp

exp refer (ref)

where exp is an expression and ref is a reference. The first form is used in most cases; it must yield a value that can be converted to a 'fixed binary(24)' value. The second form is described later in Section VII, "Storage Management."

ABBREVIATIONS AND DEFAULTS

The keyword 'dimension' can be abbreviated in two ways, as follows:

<u>Keyword</u>	<u>Abbreviation</u>
dimension	dim
dimension	(omit the entire keyword 'dimension' if it immediately follows the name of the variable being declared.)

Since recommended practice is to write the 'dimension' attribute just after the variable name, the keyword is usually omitted. For example, consider the declaration:

```
dcl C dimension(1:12,-s:1,1:3*m-2) fixed;
```

This declaration can be abbreviated as follows:

```
dcl C(1:12,-s:1,1:3*m-2) fixed;
```

there are some circumstances in which the declared name is not given in the declaration, and then either the abbreviation 'dim' must be used or the dimension must be the first attribute. Consider the following declaration:

```
dcl P entry(dim(0:5) fixed);
```

This declaration asserts that 'P' is the name of a procedure that takes one argument which must be a one-dimensional array whose subscript runs from zero to five.

The lower bound can be omitted; in that case, its default value is one. That is, if hb is any valid higher bound, then the bound-pair:

```
1:hb
```

can be written as:

```
hb
```

By means of this default, the declaration of 'C' given in the preceding paragraph can be shortened to:

```
dcl C(12,-s:1,3*m-2) fixed;
```

ARRAY STORAGE TYPES

The bound-pairs of the 'dimension' attribute in a storage type are normalized. A normalized bound-pair is:

```
1 : hb-lb+1
```

where lb and hb are the lower and upper bounds of the original, unnormalized, bound-pair. Observe that normalization of a bound-pair causes the lower extent to be '1' while leaving the difference between the lower and upper bounds unchanged.

As examples of the normalization of 'dimension' attributes, consider the arrays 'D' and 'E', declared as follows:

```
dcl D(5:n+1) float;
```

```
dcl E(2*i:2*i+3) float;
```

Suppose these arrays are allocated when $n=7$ and $i=-1$. Then the dimension attributes are:

<u>Unnormalized</u>	<u>Normalized</u>
dim(5:8)	dim(1:4)
dim(-2:1)	dim(1:4)

The unnormalized bound-pair determines the designators of the elements of each variable. The designators are:

```
D(5)    D(6)    D(7)    D(8)
E(-2)   E(-1)   E(0)    E(1)
```

The normalized bound-pair determines the storage type of each variable; therefore, both 'D' and 'E' have the same storage type, namely:

```
dim(1:4) float
```

or, using the default for the lower bound:

```
dim(4) float
```

Other examples of normalization are given in the examples of array storage that follow the next paragraph.

The elements of an array are arranged in storage in left major order. Given the designators for two elements, the order in which they appear is determined by the leftmost subscript for which the elements differ. Consider the following declaration:

```
dcl B(1:10,0:1,1:3) float;
```

The order of the elements of 'B' in storage is:

```
B(1,0,1)
B(1,0,2)
B(1,0,3)
B(1,1,1)
B(1,1,2)
B(1,1,3)
B(2,0,1)
B(2,0,2)
B(2,0,3)
B(2,1,1)
```

... (and so on for 50 more elements)

Observe that 'B(1,0,1)' precedes 'B(1,0,2)' because the first two subscripts are the same and '1' comes before '2'. Observe that 'B(1,1,2)' precedes 'B(2,0,1)' because the designators differ in their first subscript and '1' comes before '2'.

EXAMPLES

As a first example of storage for an array, consider the variable 'A1', declared as follows:

```
dcl A1(0:2) fixed dec(5);
```

The storage type for 'A1' is:

```
dim(3) fixed dec(5)
```


As a fifth and final example, consider the variable 'list', declared as follows:

```
dcl 01 list,
    02 count fixed bin,
    02 account(1000),
    03 name,
        04 last char(30) varying,
        04 middle char(1),
        04 first char(20) varying,
        04 title char(4) varying,
    03 number pic"9999",
    03 address,
        04 street char(30),
        04 loc,
            05 city char(20),
            05 state char(2),
            05 zip pic"99999",
    03 balance pic"$,,$$,,$$pv.99",
    03 credit_balance pic"$,,$$,,$$9v.99";
```

This example picks up from the last example, 'account', that was given under "Structures". The variable 'list' provides for a maximum of 1000 accounts and also provides a variable, 'list.count', to record the current number of accounts. Thus this single variable can provide the complete permanent record of the financial activity of the organization.

Guidelines for Aggregates

The choice between a structure and an array is based primarily on the following considerations:

- The data type of the members of a structure can differ from one another, whereas the elements of an array must all have the same data type.
- The selection of a member of a structure must be made when the reference is written, whereas the selection of an element of an array can be performed, by the evaluation of subscript expressions, when the array reference is evaluated.

Often arrays and structures can be combined to provide a good organization for a complicated data object.

The most important efficiency consideration for aggregates arises in connection with the bounds in the declaration of an array variable. When an array is declared with bounds that are all constants, references to the array may be an order of magnitude less expensive than when bounds are variable. When the programmer has a choice, he should use constant bounds.

When an array with a variable bound appears as a member of a structure, it should appear as late in the structure as possible. The same guideline applies to a 'character' or 'bit' string with a variable maximum length and to an 'area' with a variable area size. Consider the following example:

```
dcl 01 mem,  
    02 name(maxcnt) char(30),  
    02 cnt fixed;
```

This declaration does not conform to the guideline just given: the array with a variable bound comes first in the structure. A reference to 'name' uses the base address of the structure; that creates no problem. However, a reference to 'cnt' uses the base address of the structure plus the number of words occupied by 'name'. Because 'name' has a variable bound, the address of 'cnt' cannot be calculated by the compiler; it must be calculated each time 'cnt' is referenced during execution of the program. Now consider the following revision of the declaration:

```
dcl 01 mem,  
    02 cnt fixed bin,  
    02 name(maxcnt) char(30);
```

This declaration does conform to the guideline. A reference to 'cnt' uses the base address of the structure and a reference to 'name' uses the base address of the structure plus the amount of storage (one word) occupied by 'cnt'. Thus the addresses of both members of the structure can be calculated by the compiler.

ALIGNMENT

At the beginning of this section, it was observed that every variable has a data type, an aggregate type, and an alignment type. The data type and the aggregate type determine the values that can be accommodated by a storage unit. In contrast, the alignment type affects the way a variable is laid out in hardware storage; it has no effect on the types of values that can be accommodated by the storage unit. The alignment type is given as a single attribute, either 'aligned' or 'unaligned'.

As a simple example of alignment, consider the following declaration of an array of structures:

```
dcl 01 cell(4096),  
    02 type fixed bin(2),  
    02 cdr fixed bin(14),  
    02 flags bit(3),  
    02 car fixed bin(14);
```

Since no alignment attributes are given for the array, various alignments are assumed for the array, for each element of the array, and for each member of each element. The effect is that (for this particular example), each member occupies one word of Multics storage, each element of the array occupies four words, and the entire array occupies 4*4096 words. In this form, the array occupies more storage than necessary; however, the contents of the array can be accessed efficiently.

Now consider a slightly different declaration of the same array:

```
dcl 01 cell(4096) unaligned,  
    02 type fixed bin(2),  
    02 cdr fixed bin(14),  
    02 flags bit(3),  
    02 car fixed bin(14);
```

This array has exactly the same capacity as before, but it is laid out in a different way. According to the default rules, the 'unaligned' attribute given for 'cell' applies not only to the array but also to the elements and the members of the elements. As a result (for this example), the four members of each element are packed into a single word, so the entire array occupies just 4096 words. In this form, the array occupies much less storage, but references to its components are slower.

In most cases, the alignment of variables can be ignored. PL/I has default conventions for alignment that usually provide satisfactory results. However, there are two circumstances under which alignment must be considered:

- When the amount of storage occupied by a variable must be controlled, the alignment attribute is used. A change in alignment can cause a change of more than an order of magnitude in the storage required for a variable. In Multics, storage is more abundant than in a system that does not have virtual memory, but for very large arrays consideration of alignment can be important.
- When the layout of a variable with respect to the words, bytes, and bits of storage must be controlled, the alignment attribute is used. The proper use of alignment attributes can place a member of a structure in any given field of a hardware word. Layout is important when data is prepared for communication with facilities that are outside of PL/I.

The following discussion gives the information necessary for the use of the alignment attribute to control the amount and layout of storage for a variable. The alignment attribute is first described in a way that is independent of the Multics implementation of PL/I, and the abbreviations and defaults are given. Then the specific rules for the layout of a variable in Multics storage are given.

Alignment Attribute

The alignment attribute is one of the following keywords:

```
aligned  
unaligned
```

When a variable is 'aligned', its storage is laid out in a way that facilitates access; that is, extra storage is used, where necessary, to permit the use of fast hardware operations to fetch or store the value of the variable. When a variable is 'unaligned', its storage is laid out in a way that uses relatively little storage.

One way to facilitate access is to line up variables with word boundaries so that each variable occupies one or more full words; and that is the source of the keyword 'aligned'. One way to minimize the use of storage is to start each variable just where the last variable left off, so that, in some cases, a variable crosses word boundaries; and that is the source of the keyword 'unaligned'. The exact effect of the alignment attribute depends on a particular implementation of PL/I, and no more can be said of alignment in an implementation-independent way than what is said in the previous paragraph. Later in this section, specific rules for the layout of variables in storage are given for Multics, and these rules include the effect of the alignment attribute.

The alignment attribute is given in a 'declare' statement in the same way as other attributes; however, the alignment attribute for an array variable serves a double purpose. Consider the following examples:

```
dcl x(100) bit(1) aligned;

dcl 01 A(3) unal,
      02 m1 dec(5) unal,
      02 m2 dec(5) unal;
```

The alignment attribute on the first line of each of these statements applies both to an array and to each of its elements. Thus 'x' is an 'aligned' array, each of whose elements is an 'aligned' scalar variable; and 'A' is an 'unaligned' array, each of whose elements is an 'unaligned' structure.

Abbreviations and Defaults

The alignment attribute has just one abbreviation, but it has an elaborate default convention that permits most alignments to be handled by default. The abbreviation is:

<u>Keyword</u>	<u>Abbreviation</u>
unaligned	unal

Now consider the default convention. When the alignment of a variable is not written explicitly in the declaration of the variable, then it is determined in two steps, as follows:

1. If the variable is contained in an aggregate variable that has an explicit alignment attribute, then the variable takes its alignment attribute from the smallest containing aggregate that has an explicit alignment attribute.

2. Otherwise, the variable takes its alignment from the following default rules:

<u>Storage Type</u>	<u>Default</u>
scalar	
arithmetic	aligned
string	unaligned
address	aligned
area	aligned
aggregate	
structure	unaligned
array	(same as its elements)

The irregularity of the defaults in Step 2 reflects the intention of the designers of PL/I to provide the best choice for each case. For example, the use of 'aligned' arithmetic variables greatly increases the speed of calculations, whereas 'unaligned' string variables can save storage without much affect on the speed of access.

Some examples of this default convention follow. Consider the declaration:

```
dcl x fixed bin;
```

This declaration has no explicit alignment attribute, so one must be supplied. Step 1 of the default rule does not apply because 'x' is not contained in an aggregate. Step 2 supplies the default 'aligned'.

As a second example, consider the following declaration of a structure:

```
dcl 01 pair1 unal,  
      02 i fixed bin,  
      02 j fixed bin;
```

Step 1 of the default rule applies for both 'pair1.i' and 'pair1.j'. The result is as if the programmer had written:

```
dcl 01 pair1 unal,  
      02 i fixed bin unal,  
      02 j fixed bin unal;
```

As a third example, suppose that 'pair2' is declared in a similar way, but with no alignment attribute at all, as follows:

```
dcl 01 pair2,  
      02 i fixed bin,  
      02 j fixed bin;
```

Step 1 does not apply. According to Step 2, an equivalent declaration is;

```
dcl 01 pair2 unal,  
      02 i fixed bin aligned,  
      02 j fixed bin aligned;
```

Compare this example to the preceding example. Because 'pair2' is a structure, it is 'unaligned' by default; the two examples are the same in this respect. But the numbers 'i' and 'j' are handled differently because the alignment attribute of 'pair2' is not explicit and Step 2 applies rather than Step 1.

As a final example, consider the following declaration of a structure, which is chosen to illustrate a wide variety of defaults:

```
dcl 01 S,  
    02 s1 bit(3),  
    02 s2(500) aligned,  
    03 a pic"s99v.99",  
    03 b char(2) unal,  
    03 c,  
    04 alpha char(60),  
    04 beta fixed dec(5) unal,  
    02 s3 fixed;
```

The application of Rule 1 supplies some of the required alignment attributes, with a result that is equivalent to the following declaration:

```
dcl 01 S,  
    02 s1 bit(3),  
    02 s2(500) aligned,  
    03 a pic"s99v.99" aligned,  
    03 b char(2) unal,  
    03 c aligned,  
    04 alpha char(60) aligned,  
    04 beta fixed dec(5) unal,  
    02 s3 fixed;
```

The application of Rule 2 completes the default alignments, with a result that is equivalent to:

```
dcl 01 S unal,  
    02 s1 bit(3) unal,  
    02 s2(500) aligned,  
    03 a pic"s99v.99" aligned,  
    03 b char(60) unal,  
    03 c aligned,  
    04 alpha char(2) aligned,  
    04 beta fixed dec(5) unal,  
    02 s3 fixed aligned;
```

Storage Layout Rules for Multics

Exact rules for the layout of a variable in the 36-bit, 4-byte words of Multics memory are given here. The rules assume that the starting address of the variable is given, and define the layout that starts at that result. Thus the rules do not specify the relative positions of major variables in storage, but only the layout of the storage within the variable. The rules are given for scalar variables, then for structure variables, and finally for array variables.

STORAGE LAYOUT FOR SCALARS

To determine the storage layout for a given scalar variable at a given address, proceed as follows:

1. Begin the layout at the given address.
2. Use Table 3-1 to determine the required boundary for the variable. If the starting address is not at a boundary of the required type, then lay out filler storage up to the next boundary of the required type.
3. Use Table 3-1 to determine the minimum storage for the variable. Add the specified amount of storage to the layout.
4. If the layout does not end at a boundary of the required type (as determined in Step 2), then lay out supplementary storage up to the next boundary of the required type.

Generally, filler storage is not used in any way; in contrast, supplementary storage is used in conjunction with the minimum storage to permit a larger and more convenient representation of the stored value.

Table 3-1

Boundary and Length for Scalar Variables

Data Type	Required Boundary		Minimum Storage
	aligned	unaligned	
real fixed binary(p,q) 1 ≤ p ≤ 35 36 ≤ p ≤ 71	word even word	bit bit	(p+1) bits (p+1) bits
real fixed decimal(p,q)	word	byte	(p+1) bytes
real float binary(p) 1 ≤ p ≤ 27 28 ≤ p ≤ 63	word even word	bit bit	(p+9) bits (p+9) bits
real float decimal(p)	word	byte	(p+2) bytes
complex fixed binary(p,q)	even word	bit	2*(p+1) bits
complex fixed decimal(p,q)	word	byte	2*(p+1) bytes
complex float binary(p)	even word	bit	2*(p+9) bits
complex float decimal(p)	word	byte	2*(p+2) bytes
character (ml) nonvarying varying	word word	byte word	(ml) bytes (ml+4) bytes
bit (ml) nonvarying varying	word word	bit word	(ml) bits (ml+36) bits
picture"p" (with related data type 'char(n)')	word	byte	(n) bytes
label	even word	even word	4 words
entry	even word	even word	4 words
format	even word	even word	4 words
pointer *	even word	bit	36 bits
offset	word	word	4 words
file	even word	even word	4 words
area(as)	even word	even word	(as) words

* Note: an aligned pointer uses two full words.

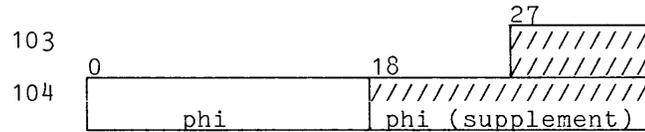
As the basis for an example of the layout of a scalar variable, consider the following declaration:

```
dcl phi fixed bin;
```

According to the default rules, this declaration is equivalent to:

```
dcl phi real fixed binary(17) aligned;
```

Suppose the last allocated bit is bit 26 of word 103. Then the rules just given prescribe the following layout for 'phi':



This layout is determined as follows:

1. The layout begins at bit 27 of word 103.
2. According to Table 3-1, the required boundary for the variable is word. Since the starting address is not at a word boundary, the layout begins with one byte of filler storage (fully shaded).
3. The layout continues with the minimum storage for the variable, 18 bits.
4. Since the required boundary is word, the layout concludes with 18 bits of supplementary storage (half shaded).

The storage available for 'phi' is a full word, the minimum plus the supplement, and the full word is used to contain the value of 'phi'. Since this eliminates masking and shifting operations, it is an important contribution to efficiency of access.

STORAGE LAYOUT FOR STRUCTURES

To determine the storage layout for a given structure variable at a given address, proceed as follows:

1. Begin the layout at the given address.
2. Determine the required boundary type for the structure as follows:
 - a. Make a list of the required boundaries for the members of the structure.
 - b. If the structure itself is 'aligned', then add the boundary word to the list.
 - c. Find the boundary on the list that refers to the largest unit of storage and take that to be the required boundary for the structure.

If the starting address is not a boundary of the required type, then lay out filler storage up to the next boundary of storage.

3. Continue the layout of the structure by laying out storage for each of its members.
4. If the required boundary is even word or word and the layout does not end at a word boundary, then lay out supplementary storage to the next word boundary.

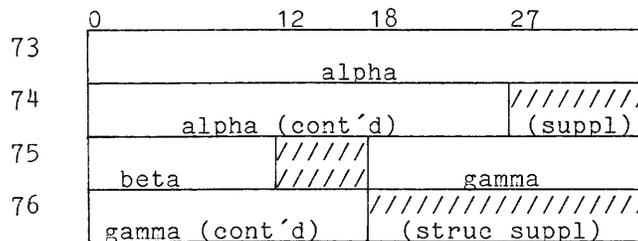
As the basis for an example of the layout of a structure variable, consider the following declaration:

```
dcl 01 s,
    02 alpha fixed dec(6,2),
    02 beta bit(12),
    02 gamma char(4);
```

According to the default rules, this declaration is equivalent to:

```
dcl 01 s unal,
    02 alpha real fixed decimal(6,2) aligned,
    02 beta bit(12) nonvarying unaligned,
    02 gamma character(4) nonvarying unaligned;
```

Suppose the starting address for the structure 's' is word 73. Then the rules prescribe the following layout for 's':



This layout is determined as follows:

1. The layout begins at bit 0 of word 73.
2. The list of required boundaries for the members of 's' is:

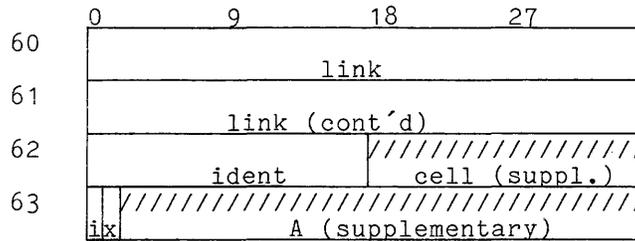
```
word
bit
byte
```

the maximal boundary from this list is word. Since the layout begins on a word boundary, no filler storage is required.

3. The layout continues with the layout of the three members of the structure. Each is laid out according to the rules for a scalar, as follows:

alpha The required boundary is word and the minimum storage is seven bytes. The layout ends with one byte of supplementary storage.

In most cases, this change in the ordering of the members of 'A' would have no affect at all on the usage of the structure, but the result is a layout that uses four words instead of seven:

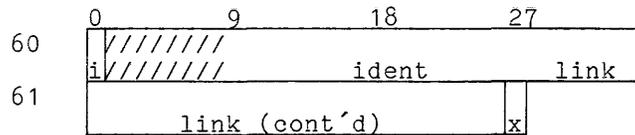


There is still some wasted storage here ('ident', 'i', and 'x' could all fit in one word), but elimination of that waste would require a change in the level structure of the variable.

Consider a different revision of the declaration of the structure 'A':

```
dcl 01 A unal,
    02 i bit,
    02 cell,
    03 ident char(2),
    03 link ptr,
    02 x bit;
```

Because of the addition of the attribute 'unal' for 'A', the layout uses two words instead of seven:



For this version, however, the interpretation of the value of the 'pointer' value will take more time than for the 'aligned' value.

STORAGE LAYOUT FOR ARRAYS

To determine the storage layout for a given array variable at a given address, proceed as follows:

1. Begin the layout at the given address.
2. The required boundary for the array is the same as for the elements of the array. If the starting address is not a boundary of the required type, then lay out filler storage up to the next boundary of the required type.
3. Continue the layout of the array by laying out storage for each of its elements.
4. If the last element does not end at the required boundary for the array, then lay out supplementary storage to the next boundary of the required type.

The alignment attribute of an array is especially important, since it is in the layout of large arrays that the alignment can have a significant effect on storage requirements. As a simple illustration, consider the following declarations:

```

dcl pm1(50,50) bit;
dcl pm2(50,50) bit aligned;

```

The array 'pm1' requires 70 words, whereas 'pm2' requires 2500 words.

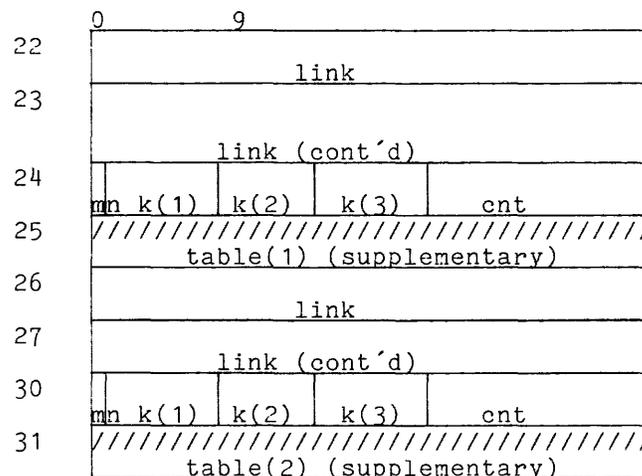
For a second example of the layout of an array, consider the following declaration:

```

dcl 01 table(1000),
    02 link ptr,
    02 cont unal,
    03 m bit,
    03 n bit,
    03 k(3) fixed(5),
    03 cnt fixed(15);

```

The layout for 'table' is as follows:



... (and so on for 998 more elements)

Observe that an element of 'table' has a required boundary of even word (because a 'pointer' variable is a part of each element. However, according to the rules for laying out a structure, an element of 'table' occupies only three words. Thus there is a word of filler storage between one element and the next. In accordance with Step 4 of the layout rules for an array, a word of filler storage follows the three words of the last element. Thus the array occupies 4000 words.

SECTION IV

VALUE CONVERSION

In some cases, a value cannot be placed in a storage unit as it is; instead, it must be represented in a form specified by the storage type of the storage unit. This adjustment of a value to a specified representation is the conversion of the value. The conversion operation is not necessarily simple; it may require that the given value be reinterpreted, approximated, truncated, or even rejected as unsuitable.

There are many storage types in PL/I; and if conversion is allowed from any type to any other type, many kinds of conversion are required. PL/I allows conversion between any types of computational values, including some conversions that are not entirely obvious, such as that of an arithmetic value into a character-string value. On the other hand, PL/I does not allow most conversions that involve non-computational values; in fact, the only conversion of this kind allowed is that between the two types of locator values, 'pointer' and 'offset'.

The storage type consists of an aggregate type as well as a data type. PL/I allows some conversions of aggregate type. For example, if a scalar value is assigned to an array variable, the value is automatically promoted to an array value before the assignment.

CONTEXTS THAT FORCE CONVERSION

In PL/I, if a conversion from one storage type to another is required but is not explicitly indicated, the conversion is implicitly performed. This implicit conversion makes a program look simpler than it really is, by concealing costly conversion operations, and it has both advantages and disadvantages. Certainly any means to shorten a program is welcome, but unpleasant surprises can occur when a programmer misunderstands the rules for implicit conversion.

The conversion of a value is forced when it is assigned to a storage unit. The storage unit to which a value is assigned is the target, and the storage type of the target determines the kind of conversion performed. Sometimes the target is a variable name that has been explicitly declared in the program; in this case, its storage type is easily determined. In other cases, the target is a temporary storage unit provided by PL/I; and, in this case, the storage type is determined by special rules. In all cases the storage type of the target depends on the context of the computation. The ways in which context forces conversion are described at many places in this manual because they occur in many features of the language. An informal review of these contexts is given in the following paragraphs in order to show clearly the role of conversion in PL/I.

General Contexts

There are a variety of contexts in which an expression is used in a general way, and in these contexts, the storage type of the target is virtually unrestricted. These contexts are discussed in the following paragraphs.

ASSIGNMENT STATEMENTS

The most fundamental context for conversion is the assignment statement. For example, the statement:

```
x = y;
```

can imply any of the possible conversions of PL/I, depending on the storage type of the variables 'x' and 'y'. The conversion can be a simple matter, as in the case:

```
dcl x fixed decimal(8,2);
dcl y fixed decimal(7,2);
...
x = y;
```

Here, the only difference is that the target has one more high-order digit than the assigned value; and the conversion is always exact conversion. On the other hand, the conversion can be quite complicated, as in the case:

```
dcl x(2,3) fixed decimal(8,2);
dcl y float binary(25);
...
x = y;
```

Here the scale attribute must be converted from 'float' to 'fixed', the base attribute from 'binary' to 'decimal', the precision attribute from '(25)' to '(8,2)', and, most remarkable, the aggregate type from scalar to that for a two-dimensional array. Thus a small assignment statement can invoke a large conversion effort. A description of the assignment statement is given in Section X, "Value Assignment."

ASSIGNMENT-LIKE CONSTRUCTS

In several important contexts, an assignment statement is implied in a more or less obvious way. For example, the 'do' group:

```
do i = 1 to j;  
  Q(i) = R(i);  
end;
```

is equivalent to:

```
      i = 1;  
loop:  if i <= j  
      then do;  
        Q(i) = R(i);  
        i = i + 1;  
        goto loop;  
      end;
```

Thus the example 'do' statement has within it the equivalent of an assignment statement and, therefore, a conversion occurs if the storage type of 'i' is not the same as that of 'j'. A full description is given in Section XI, "Program Flow."

There are a few other constructs that are closely related to the assignment statement. An 'initial' attribute assigns a value to a variable when the variable is allocated, as described in Section VII, "Storage Management." A 'refer' option, which is used in the declaration of some 'based' variables, assigns a value to a member of a structure in which it appears when that structure is allocated, as also described in Section VII, "Storage Management." Finally a 'get' statement assigns an input value that is a character string to an input variable, as described in Section XIV, "Stream Input/Output."

ARGUMENTS AND RESULTS

There are implied conversions in connection with the invocation of a procedure. For example, consider the following procedure:

```
F3:  proc(a) returns(fixed binary(20));  
      dcl (a,b) float;  
      b = a**3;  
      return(b);  
end;
```

Suppose this procedure is invoked by the function reference

```
F3(4.58)
```

The argument of this function reference is the literal constant '4.58' whose storage type is 'fixed decimal(3,2)', and so it must be converted before being assigned to the parameter 'a' whose storage type is 'float'. Similarly, the returned value 'b' is 'float', and so it must be converted to 'float' before being assigned to the temporary storage for the value of the function reference whose storage type is 'fixed binary(20)'. Thus conversion occurs for both argument and result in this case. A description is given in Section XII, "Procedure Invocation."

Just as conversion may be required for the invocation of a programmer-defined procedure, so also may conversion be required for a built-in function reference. However, the built-in functions do not restrict an argument to a single storage type but rather accept any one of a specified set storage types. Thus the built-in functions are designed to minimize the need for conversion of values. Details are given in Sections VIII and IX, "Expressions" and "Operations," respectively.

There are built-in functions whose purpose is the explicit conversion of a value from one storage type to another. Consider the following program fragment:

```

dcl s char(10);
dcl i fixed;
...
s = char(i,10);

```

Here the built-in function 'char' is used to convert the arithmetic value of 'i' to a character string for assignment to 's'. Full details on the built-in conversion functions are given under "Conversion Operations" in Section IX, "Operations."

Arithmetic operators are similar to built-in functions in their use of conversion. Although the operator in an expression may require conversion of its operands, the requirement is reduced in certain ways. For example, if both of the variables of the expression 'a+b' are 'real fixed decimal', then both values are used without conversion, even though they may not have the same 'precision' attribute. Details are given in Sections VIII and IX, "Expressions" and "Operations," respectively.

Special Contexts

There are many contexts in which the value of an expression is required for a special purpose, and in these contexts the value is converted to a specific storage type. For example, consider the assignment statement:

```
X(2*i+k) = 0;
```

In this statement the expression '2*i+k' appears in a context that requires a subscript; accordingly, its value is converted to a binary integer. As a second example, consider the output statement:

```
put list(alpha);
```

In this statement, the expression 'alpha' appears in a context that requires a value that can be placed in the output stream; accordingly, it is converted to a character string.

The distinctive characteristic of a special context is that the storage type of the target is implicit; that is, it is implied by the use to which the given value is put rather than being given by means of a declaration. In order to provide an overall view, the special contexts are listed here; they are classified according to the storage types of the implied targets: integer, character string, bit string, and locator.

INTEGER TARGETS

When a special context requires that the value of a given expression be converted to an integer, the target storage type is:

real fixed binary(p,0)

The number-of-digits, p, is always sufficiently large to accommodate any value that is valid in the given context. The integer contexts are as follows:

- The maximum-length expression in a 'character' or 'bit' attribute, as described in Section III, "Value Storage"
- The area-size expression in an 'area' attribute, as described in Section III, "Value Storage"
- Each of the array-bound expressions in a 'dimension' attribute, as described in Section III, "Value Storage"
- The expression in a 'position' attribute, as described in Section VII, "Storage Management"
- Each replication-factor expression in an 'initial' attribute, as described in Section VII, "Storage Management"
- Each subscript expression in a subscripted variable reference, as described in Section VIII, "Expressions"
- The expression in a 'pagesize' or 'linesize' option in an 'open' statement for a 'stream' file, as described in Section XIV, "Stream Input/Output"
- The expression in a 'skip' or 'line' option in a 'get' or 'put' statement, as described in Section XIV, "Stream Input/Output"
- Each expression in a format specification list, as described in Section XIV, "Stream Input/Output"
- The expression in an 'ignore' option in a 'read' statement, as described in Section XV, "Record Input/Output"
- The arguments of some built-in functions (for example, the second and third arguments of 'substr'), as described in Section IX, "Operations"

CHARACTER-STRING TARGETS

When a special context requires that the value of a given expression be converted to a character string, the target storage type is:

character(m1)

The maximum-length, ml, is determined from the storage type of the given expression. The character-string contexts are as follows:

- The expression in a 'title' option, as described in Sections XIV and XV, "Stream Input/Output" and "Record Input/Output," respectively
- Each expression that supplies an output value in a 'put' statement, as described in Section XIV, "Stream Input/Output"
- The expression in a 'string' option in a 'get' statement, as described in Section XIV, "Stream Input/Output"
- The expression in a 'key' or 'keyfrom' option, as described in Section XV, "Record Input/Output"

BIT-STRING TARGETS

When a special context requires that the value of a given expression be converted to a bit string, the target storage type is:

bit(ml)

The maximum length, ml, is determined from the storage type of the given expression. There are two bit-string contexts, as follows:

- The test expression in an 'if' statement, as described in Section XI, "Program Flow"
- The test expression in a 'while' option in a 'do' statement as described in Section XI, "Program Flow"

LOCATOR TARGETS

In one special context, the value of an expression must be converted to a locator target. In this case, the target storage type is:

pointer

and the given expression must already be 'pointer' or 'offset'. The locator context is:

- The locator qualifier expression in a locator qualified reference, as described in Section VIII, "Expressions"

DATA TYPE CONVERSION

Rules for all possible conversions of the data type of a scalar are given in the following paragraphs. Arithmetic, character string, bit string, and locator targets are considered.

Pictured strings are not discussed, either as targets or as the source of the value to be considered. This omission reflects the fact that when a pictured-string storage unit is accessed, it is always considered to have an arithmetic or ordinary character-string value. Section III, "Value Storage," gives the relation between a pictured storage unit and its arithmetic or character string value.

When a conversion is attempted that could be an error, an appropriate PL/I condition occurs. The occurrence of such conditions are mentioned in the following rules, but a discussion of their significance is deferred until later in this section, under the heading "Conditions for Conversions".

Arithmetic Targets

The rules for converting a scalar computational value to a given arithmetic data type follow. This is the most important kind of conversion because it includes the conversion from one type of arithmetic value to another. The conversion is performed in four steps, as follows:

1. Reinterpretation. An arithmetic value is obtained from the given value as follows:
 - a. If the given value is arithmetic, then no reinterpretation is required.
 - b. If the given value is a character-string, then it is reinterpreted as an arithmetic value. The string must have one of the following forms:

sign arithmetic-constant
sign arithmetic-constant sign arithmetic constant i
null-string

The first form represents a real value, the second a complex value, and the third a zero real value. The initial sign can be omitted if it is plus. Arithmetic constants are defined in Section VIII, "Expressions". Blanks can occur in the character string only before or after the entire value representation, not within it. If the character string does not satisfy all of these conditions, it cannot be interpreted as an arithmetic value and the 'conversion' condition occurs.

- c. If the given value is a bit string then it is reinterpreted as an arithmetic value by treating the sequence of bits as the binary representation of a positive integer, with the rightmost bit taking the units position.

2. Representation. The value is represented according to the scale and base attributes of the target data type, but with the mode and precision required by the value itself. The specific rules are as follows:
 - a. The mode is 'complex' or 'real' depending on whether the value does or does not have an imaginary part. If the value has only an imaginary part, zero is assumed for the real part.
 - b. The scale and base are those of the target data type.
 - c. The precision (number of digits and scale factor) is whatever is necessary to represent the given value. Some values cannot be represented exactly, and for such values low-order digits that do not affect the final result (obtained after the "Approximation" step) are omitted.
3. Approximation. If necessary, the value is approximated. The operation depends on the mode, scale, and precision of the target data type, as follows:
 - a. A 'fixed' value representation is adjusted to have the number of fractional digits that is specified by the target precision. This may require the addition of zeros at the right or the truncation of low-order digits. No rounding occurs.
 - b. A 'float' value representation is adjusted to have the number of mantissa digits that is specified by the target precision. This may require the addition of zeros or the truncation of low-order digits. Rounding occurs in the case of 'float' values.
 - c. A 'complex' value representation is adjusted to the target mode. If the target mode is 'real', then this requires the truncation of the imaginary part of the value representation.
4. Range Check. The magnitude of the value is checked to determine whether or not it can be accommodated by the target data type. The following cases apply:
 - a. A 'fixed' value representation is adjusted to have the high-order digits that are specified by the target precision. This may require the addition of zeros on the left or the truncation of high-order digits. If truncation of nonzero high-order digits occurs, then the 'size' or 'fixedoverflow' condition occurs.
 - b. A 'float' value representation has its exponent checked. If the exponent is greater than 127, the 'overflow' condition occurs. If the exponent is less than -128, the 'underflow' condition occurs.

This concludes the rules for conversion to an arithmetic target. A good way to learn such a set of rules is to extract from them those features that are not obvious. What is "not obvious" depends on the reader, but for the rules just given, the following items might be selected:

- Blanks can appear before or after but not within a character-string representation of an arithmetic value.

- A character string that is all blanks is interpreted as the arithmetic value zero.
- When an arithmetic value is approximated, rounding occurs if the target is 'float'.
- When a complex value is converted to 'real', the imaginary part is discarded without the occurrence of a condition to report a possible error.

EXAMPLES OF ARITHMETIC TO ARITHMETIC CONVERSION

As a first example, suppose the given value is the character string "25.97181000" and the target data type is 'real float decimal(5)'. The first step in the conversion is the reinterpretation of the string value to yield an arithmetic value, giving 25.97181. The second step is the representation of the value according to the target scale and base attributes, which are 'float' and 'decimal', giving '+2597181e-5'. The third step is the approximation of the value as specified by the target precision, which is '5', giving '+25972e-3'. The fourth step is the range check which, in this case, determines that the exponent is in the required range of -128 through +127. The final result of the conversion, '+25972e-3' is a valid value representation for the given data type.

The following examples show the conversion of an arithmetic value for an arithmetic target:

<u>Given Value</u>	<u>Target Type</u>	<u>Exact Representation</u>	<u>Result Value</u>
17.876	fixed dec(5)	+17.876	+00017.
17.876	fixed dec(5,1)	+17.876	+0017.8
17.876	fixed dec(5,3)	+17.876	+17.876
17.876	fixed dec(5,4)	+17.876	(size)
131	fixed bin(12)	+10000011.b	+000010000011.b
131	fixed bin(12,2)	+10000011.b	+0010000011.00b
131	fixed bin(12,5)	+10000011.b	(size)
-6.9	fixed bin(9,6)	-110.1110011...b	-110.111001b
-6.9	fixed bin(9)	-110.1...b	-000000110.b
5.638	float dec(5)	+5638.e-3	+56380.e-4
5.638	float dec(3)	+5638.e-3	+564.e-2

The first group of examples (for the given value 17.876) show the effect of gradually increasing the scale factor of the target until, finally, there are not enough integer digits and the 'size' condition occurs. The second group is a similar sequence for a binary target. The third group shows the handling of a value, -6.9, that cannot be expressed exactly in binary representation; the "..." in the exact representation means "digits that do not affect the final result". The last group shows a 'decimal float' target; and includes the only example in which rounding occurs when low-order digits are truncated.

EXAMPLES OF STRING TO ARITHMETIC CONVERSION

The following examples illustrate the conversion of character-string or bit-string values to arithmetic targets.

<u>Given Value</u>	<u>Target Type</u>	<u>Reinterpretation</u>	<u>Result Value</u>
"-8.92" " " " "	fixed dec(5)	-8.92	-00008.
"-8.92" " " " "	fixed dec(5)	(conversion)	
" " " " " "	fixed dec(5,2)	0	+000.00
" " " " " "	fixed dec(5,2)	0	+000.00
23i	complex dec float(5)	0+23i	+00000.e0+23000.e-3i
23i	dec float(5)	0+23i	+00000.e0
"8.23e-2"	fixed dec(5,2)	.0823	+000.08
"8.23-2"	fixed dec(5,2)	(conversion)	
"8.23e-127"	float dec(3)	823e-129	(underflow)
"101"b	fixed dec(4,1)	5	+005.0
" "b	fixed dec(4,1)	0	+000.0
"0000000"b	fixed dec(4,1)	0	+000.0

The first group of examples shows ways in which blanks can and cannot be used in a character string that is to be converted into an arithmetic value. The second group shows how a complex value is filled out or truncated as the target requires. The third group shows how the wrong format for a floating-point value causes the 'conversion' condition to occur and also shows how the conversion of a number that appears to be in range can cause the 'underflow' condition to occur. The final group shows the conversion of bit-string values to arithmetic values.

Character-String Targets

The conversion of a scalar computational value to a character-string value is performed according to one of the following three rules:

1. Given Character-String. Suppose the given value is a character string. Let the string type of the target be 'character(n)', where n is the maximum length. Then the conversion is as follows:
 - a. If the length of the given value is not greater than n and the target is 'varying', then the given value (with its given length) is the result.
 - b. If the length of the given value is not greater than n, as before, but the target is 'nonvarying', then blanks are added at the right end of the given string until it is n characters long. The extended value is the result.
 - c. If the length of the given value is greater than n, then the 'stringsize' condition occurs.
2. Given Bit-String. Suppose the given value is a bit-string. It is reinterpreted as a character-string value by interpreting each bit as a '0' or '1' character. The resulting character-string value is then adjusted to the correct length by Rule 1, above.

3. Given Arithmetic Value. Suppose the given value is an arithmetic value. Then the following steps are performed:

- a. This step depends on scaling and base of the given value (not the target). A new number-of-digits, pc, and scale-factor, qc, must be defined. If the given value is 'decimal(p,q)', then no conversion is required, and:

$$\begin{aligned}\underline{pc} &= p \\ \underline{qc} &= q\end{aligned}$$

If the given value is 'fixed binary(p,q)', then the given value is converted to 'fixed decimal(pc,qc)' where

$$\begin{aligned}pc &= \min(\text{ceil}(p/3.32)+1, 59) \\ qc &= \text{ceil}(q/3.32) \quad (\text{for } q \text{ not negative}) \\ qc &= -\text{ceil}(-q/3.32) \quad (\text{for } q \text{ negative})\end{aligned}$$

If the given value is 'float binary(p)', then the given value is converted to 'float decimal(pc)' where

$$pc = \min(\text{ceil}(p/3.32), 59)$$

Both of these conversions are performed according to the rules for an arithmetic target given earlier in this section. At the conclusion of this step, the given value necessarily has base 'decimal'. The mode of the value is not changed by this step.

- b. Next, the value is expressed as a character string. Observe that the representation is different from that used for the value when it is in a storage unit.

(1) If the value is 'real float', it is expressed as follows: a blank (for plus) or a '-', followed by the first digit of the mantissa, followed by the decimal point, followed by the remaining digits of the mantissa, followed by 'e' followed by a signed, three-digit exponent. Let pc be the number-of-digits specified by the precision of the given value. Then the mantissa has pc digits and the whole character string has length pc+7.

(2) If the value is 'real fixed', the rules for its representation as a character-string are complicated and are best given by examples (see below). Let pc and qc be the number-of-digits and scale factor specified by the precision of the given value.

When $\underline{pc} \geq \underline{qc} \geq 0$ (the usual case), the character string has length pc + 3. Otherwise (i.e., $\underline{qc} > \underline{pc}$ or $\underline{qc} < 0$), the character string has length pc + 3 +k, where k is the number of characters required to represent the absolute value of qc without leading zeros.

(3) If the value is 'complex', its representation is obtained by writing the two parts as 'real' values according to rule (1) or (2), just given, and then concatenating them. If the imaginary part is positive, the blank that indicates its sign is changed to '+'. An 'i' is added just after the imaginary part. Blanks that occur between the two parts are removed and are placed at the right end of the representation. Let lr and li be the lengths of the strings representing the real and imaginary parts. Then the entire character string has the length lr+li+1.

- c. The character string that results from Step b is adjusted to the correct length by Rule 1, above.

This concludes the rules for conversion to a character-string target. These rules are used chiefly in connection with data-directed and list-directed output, where they determine the format of the values in the listing. The rules are complicated, especially when the given value is binary. The following table will be useful in determining the precision of a binary value converted to a decimal value under Step 3.a:

<u>n</u>	<u>ceil(n/3.32)</u>	<u>n</u>	<u>ceil(n/3.32)</u>
1-3	1	37-39	12
4-6	2	40-43	13
7-9	3	44-46	14
10-13	4	47-49	15
14-16	5	50-53	16
17-19	6	54-56	17
20-23	7	57-59	18
24-26	8	60-63	19
27-29	9	64-66	20
30-33	10	67-69	21
34-36	11	70-73	22

This table includes the maximum number-of-digits for a Multics PL/I binary value (71).

EXAMPLES OF CHARACTER-STRING TO CHARACTER-STRING CONVERSION

The following examples illustrate conversion of character-string values to character-string targets.

<u>Given Value</u>	<u>Target Type</u>	<u>Result Value</u>
"ABC"	char(6) varying	"ABC"
"ABC"	char(6)	"ABCØØØ"
"ØABC"	char(6)	"ØABCØØ"
"ABCØØ"	char(6)	"ABCØØØ"
""	char(6)	"ØØØØØØ"
"ABCDEFG"	char(6)	(stringsize)

When the 'varying' attribute is present, the length of the given value is not changed. When the 'varying' attribute is not present, 'nonvarying' is assumed and the given value is extended to the maximum length. The examples show that blanks that are already in the given string are treated as ordinary characters. When the length of the given string exceeds the maximum length allowed by the target, the 'stringsize' condition occurs.

EXAMPLES OF BIT-STRING TO CHARACTER-STRING CONVERSION

The following examples illustrate conversion of bit-string values to character-string targets.

<u>Given Value</u>	<u>Target Type</u>	<u>Result Value</u>
"0100"b	char(7) varying	"0100"
"0100"b	char(7)	"0100ØØØ"

Observe that extension of the string occurs after its conversion to a character string, so the added characters are blanks, not zeros.

EXAMPLES OF ARITHMETIC TO CHARACTER-STRING CONVERSION

Many examples are required to adequately illustrate the conversion of arithmetic values to character-string targets. Groups of examples will be given for each of the following types of given arithmetic values:

```
float decimal
fixed decimal
float binary
fixed binary
complex
```

The conversions can be understood in two ways. In many cases, it is enough to determine how many characters the character-string form of an arithmetic value requires. In some cases it is necessary to know the exact representation that is used. The examples illustrate both levels of knowledge.

Float Decimal Values

The following examples show the conversion of 'float decimal' values to character-string targets.

<u>Given Type</u>	<u>Given Value</u>	<u>Target Type</u>	<u>Result Value</u>
float dec(5)	-81993.e6	char(14)	"-8.1993e+010" (string size)
float dec(5)	+81993.e6	char(14)	"8.1993e+010" (string size)
float dec(5)	+81993.e6	char(10)	"81993e6" (string size)

In the first two examples,

```
p = 5          (number-of-digits in given value)
p+7 = 12      (length of the representation)
length = 14   (length required by the target)
```

Observe that three digits are always provided for the exponent; this is part of a design policy that assures that the representation always takes $p+7$ columns, regardless of the particular given value.

Fixed Decimal Values

The following examples illustrate conversion of 'fixed decimal' values to character-string targets.

<u>Given Type</u>	<u>Given Value</u>	<u>Target Type</u>	<u>Result Value</u>
fixed dec(4,2)	-49.62	char(20) var	"-49.62" (string size)
fixed dec(4,2)	-00.02	char(20) var	"-00.02" (string size)
fixed dec(4,2)	+00.02	char(20) var	"00.02" (string size)
fixed dec(4,0)	-8200.	char(20) var	"-8200." (string size)
fixed dec(4,0)	+0000.	char(20) var	"0000." (string size)

Because the target is 'varying', the result string is not extended to 20 characters. Observe that the length of the result is always $p+3$ (where p is the number-of-digits of the given value); that is, although leading zeros, the plus sign, and a trailing decimal point are all suppressed, blanks are added at the left to compensate for the suppressed characters. Thus it is easy to calculate the length of the result.

When a value is 'fixed decimal(p,q)' and the scale factor is negative ($q < 0$) or the scale factor is greater than the number-of-digits ($q > p$), then the decimal point is not adjacent to a variable digit of the value; instead it is adjacent to a sequence of filler zeros, as the examples below show. Such fixed-point values are given special treatment, as follows:

<u>Given Type</u>	<u>Given Value</u>	<u>Target Type</u>	<u>Result Value</u>
fixed dec(4,-2)	+066700.	char(20) varying	"00667f+2 "
fixed dec(7,11)	-.00008131885	char(20) varying	"-8131885f-11 "
fixed dec(7,11)	-.00000009487	char(20) varying	"000-9487f-11 "

The slashed zeros are the filler zeros. In the representation of these values no attempt is made to place the decimal in the representation of the value; instead, each value is expressed as an integer with a scaling factor. The scaling factor in the representation is the negative of the scaling factor in the precision attribute.

Float Binary Values

The following examples illustrate conversion from 'float binary' values to character-string targets. The target type 'char(20) varying' is assumed in each case.

<u>Given Type</u>	<u>Given Value</u>	<u>Intermediate Value</u>	<u>Result Value</u>
float bin(27)	+.1011001 ... e7b	+890000000.e-7	"08.90000000e+001"
float bin(20)	-.1000000 ... e-9b	-9765625.e-10	"-9.765625e-004"

The '...' indicates zero digits that fill out the required precision of the binary value. The "intermediate value" is the value after it has been converted to a 'decimal' value but before it has been converted to a character string. The data type for the intermediate value is calculated according to Rule 3.a. The calculation is as follows:

First Example

$p = 27$
 $p/3.32 = 8.1 \dots$
 $\text{ceil}(p/3.32) = 9$

Second Example

$p = 20$
 $p/3.32 = 6.02 \dots$
 $\text{ceil}(p/3.32) = 7$

The intermediate data types for the two examples are 'float dec(9)' and 'float dec(7)'.

Fixed Binary Values

The following examples illustrate the conversion of 'fixed binary' values into character-string targets. The target is assumed to be 'char(20) varying'.

<u>Given Type</u>	<u>Given Value</u>	<u>Intermediate Value</u>	<u>Result Value</u>
fixed bin(9,0)	-001010101.b	-0085	"ØØØØ-85"
fixed bin(9,0)	-111111111.b	-0511.	"ØØØØ-511"

The "intermediate value" is again the value after it has been converted to a 'decimal value' and before it has been converted to a character string. The data type for the intermediate value is calculated as follows:

<u>Number-of-digits</u>	<u>Scale-factor</u>
p = 9	q = 0
p/3.32 = 2.7 ...	q/3.32 = 0
ceil(p/3.32) = 3	ceil(p/3.32) = 0
ceil(p/3.32)+1 = 4	

Thus the intermediate type is 'fixed dec(4,0)'. The second example shows that the formulae allowed one more digit (4 instead of 3) than were required by the largest possible value of the given data type. The example in the next paragraph shows the reason for this aspect of the design of PL/I.

The following is one more example of the conversion of a 'fixed binary' value to a character-string target.

<u>Given Type</u>	<u>Given Value</u>	<u>Intermediate Value</u>	<u>Result Value</u>
fixed bin(9,1)	-11111111.1b	-255.5	"Ø-255.5 "

Here the data type of the intermediate value is calculated to be 'fixed dec(4,1)'. The full four digits of precision are required to represent the value '-127.5'. It thus becomes apparent that although a nine-bit integer never needs four decimal digits, a nine-bit number may need four decimal digits. In any case, the Rule 3.a always applies.

Complex Values

The following example illustrates the conversion of complex values into character-string targets.

<u>given type:</u>	complex float bin(20)
<u>given value:</u>	-.10000000000000000000e-1b+.11100000000000000000e1bi
<u>intermediate type:</u>	complex float dec(7)
<u>intermediate value:</u>	-2500000e-6+1750000e-6i
<u>target type:</u>	char(32)
<u>result value:</u>	"-2.500000e-001+1.750000e+000iØØØ"

Bit-String Targets

The conversion of a scalar computational value to a bit-string value is performed according to one of the following three rules:

1. Given Bit-String. The rules for adjusting the length of a bit-string to suit the target are the same as for character-string to character-string conversion with one exception: whereas a character string is extended with blank characters, a bit string is extended with zero-bits.
2. Given Character-String. Suppose the given value is a character-string. If every character is a '0' or a '1' character, then the character-string is reinterpreted as a bit-string by interpreting each character as a bit. The resulting bit-string is then converted to a bit-string of the required length by Rule 1, above. If the character string contains a character other than '0' or '1', the 'conversion' condition occurs.
3. Given Arithmetic Value. Suppose the given value is arithmetic. First, the value pc is computed according to the following table:

<u>Attributes of Given</u>	<u>Value of 'pc'</u>
fixed binary(p,q)	min(71,max(p-q,0))
fixed decimal(p,q)	min(71,max(ceil((p-q)*3.32),0))
float binary(p)	min(71,p)
float decimal(p)	min(71,ceil(p*3.32))

If the value of pc is zero, then the null bit string is immediately adopted as the result of the conversion. Otherwise, the given value is converted to an intermediate value of data type 'real fixed binary(pc,0)'. The result of this conversion is a 'binary' integer value. This value is reinterpreted as a bit-string by ignoring the sign and treating each binary digit as a bit. The resulting bit string is adjusted to the correct length by Rule 1, above.

EXAMPLES OF BIT-STRING TO BIT-STRING CONVERSION

The following examples illustrate the conversion of bit-string values to bit-string targets.

<u>Given Value</u>	<u>Target Type</u>	<u>Result Value</u>
"11010"b	bit(8) varying	"11010"b
"11010"b	bit(8)	"11010000"b
""b	bit(4) varying	""b
""b	bit(4)	"0000"b
"11010"b	bit(4)	(stringsize)

The examples show the difference between conversion to a 'varying' or a 'nonvarying' target. The last example shows that the 'stringsize' condition occurs when the given string is too long for the target.

EXAMPLES OF CHARACTER-STRING TO BIT-STRING CONVERSION

The following examples illustrate the conversion of character-string values to bit-string targets.

<u>Given Value</u>	<u>Target Type</u>	<u>Result Value</u>
"11010"	bit(8) var	"11010"b
"11010"	bit(8)	"11010000"b
""	bit(4) var	""b
""	bit(4)	"0000"b
"11010"	bit(4)	(stringsize)
"0120"	bit(4)	(conversion)
"101Ø"	bit(4)	(conversion)

The first five examples parallel the examples given for bit-string to bit-string conversion in the previous paragraph. The last two examples show that if a character that is not '0' or '1' appears in the given string, then the 'conversion' condition occurs.

EXAMPLES OF ARITHMETIC TO BIT-STRING CONVERSION

Fixed Binary Values

The following examples illustrate conversion of 'fixed binary' values to bit-string targets. In each example, the target is assumed to be 'bit(20) varying'.

<u>Given Type</u>	<u>Given Value</u>	<u>pc</u>	<u>Intermediate Value</u>	<u>Result Value</u>
fixed bin(5,0)	-01011.b	5	-01011.b	"01011"b
fixed bin(5,3)	+10.111b	2	+10.b	"10"b
fixed bin(5,-3)	+10111ØØØ.b	8	+10111000.b	"10111000"b
fixed bin(5,6)	+.Ø10111b	0	(not required)	""b

The "intermediate value" is the value after it has been converted to a 'fixed binary(pc,0)' data type. In the last example, pc is zero, so the null bit-string is assumed without resort to an intermediate value. In all the examples, the result bit-string is a representation of the integer digits of the given value.

Fixed Decimal Values

The following examples illustrate conversion of 'fixed decimal' values to bit-string targets. In each example, the target is assumed to be 'bit(20) varying'.

<u>Given Type</u>	<u>Given Value</u>	<u>pc</u>	<u>Intermediate Value</u>	<u>Result Value</u>
fixed dec(5,2)	-034.95	10	-0000100010.b	"0000100010"b
fixed dec(5,6)	+.Ø81327	0	(not required)	""b

The value of pc for the first example is calculated from a formula in Rule 3, above, as follows:

```
p = 5, q = 2
p-q = 3
(p-q)*3.32 = 3*3.32 = 9.96
ceil((p-q)*3.32) = ceil(9.96) = 10
max(ceil(p-q)*3.32,0) = max(10,0) = 10
min(71,max(ceil(p-q)*3.32,0)) = min(71,10) = 10
```

Thus the data type of the intermediate value for the first example is 'fixed bin(10,0)'. In the second example, pc comes out to be zero so that the null bit-string is the result without resort to an intermediate value.

Float Values

The following examples illustrate conversion of 'float' values to bit-string targets. In each example, the target is assumed to be 'bit(20) varying'.

<u>Given Type</u>	<u>Given Value</u>	<u>pc</u>	<u>Intermediate Value</u>	<u>Result Value</u>
float bin(10)	-.1101100100e+6b	10	-0000110110.b	"0000110110"b
float dec(2)	+59e-1	7	+0000101.b	"0000101"b

Complex Values

An example of a 'complex' given value is not included here because the effect of the conversion of the value to an intermediate 'fixed bin(pc,0)' is to discard its imaginary part and treat it as a 'real' value.

Locator Targets

There are two kinds of locator values: pointer and offset. Conversion between the two kinds of locator occurs without any restrictions or special rules. For example, in the statement

```
cur->cell = alpha;
```

the variable 'cur' appears as a locator qualifier and must have a locator value. If 'cur' is a 'pointer' variable, its value is used as is; but if it is an 'offset' variable, its value is automatically converted to a 'pointer' value.

AGGREGATE TYPE CONVERSION

All possible conversions of aggregate types are described in this section. The target aggregate type is always known when a conversion is performed. Conversion is allowed only in certain simple cases in which the aggregate type of the given value is the aggregate type of a component of the target. In some descriptions of PL/I, the conversion of an aggregate type is referred to by a special word "promotion".

The rules for converting a value to a given aggregate type follow. There are only two cases for which aggregate type conversion can occur, as follows:

1. Scalars. A scalar can be converted into any aggregate by taking the value of the scalar to be the value of each scalar component of the aggregate.
2. Structures. A structure can be converted into an array whose elements are structures of the same aggregate type as the given structure. The value of the given structure is taken to be the value of each element of the array.

When a given value differs from an aggregate target and cannot be converted by the rules just given, the program is invalid.

When the aggregate type of the given value agrees with that of the target (either from the beginning or after conversion), then each component of the given aggregate must be converted to the data type of the corresponding component of the target. The conversion of the components proceeds according to the rules already given for type conversion. Each such conversion is a distinct operation, and the conversions do not occur in any defined order.

Example of Aggregate Conversion

The following example illustrates the complete conversion of a value of one aggregate type to a target of a different aggregate type.

```
given type:    01, 02 fixed dec(4), 02 char(10) var
given value:  +0132., "HENRY"
target type:  01 dimension(1:2), 02 fixed bin(10), 02 char(8)

result value: +0010000100.b, "HENRYØØØØ", +0010000100.b, "HENRYØØØØ"
```

There are three distinct actions associated with this conversion:

- The aggregate type is promoted from '01,02,02' (a structure with two scalar components) to '01 dim(1:2),02,02' (an array of two structures that each have two scalar components).
- A 'fixed dec(4)' value is converted to a 'fixed bin(10)' target.
- A 'char(10) var' value is converted to a 'char(8)' target.

There is no assurance that these steps will occur in this or any other particular order; thus if a condition occurs during the conversions it is not possible to know exactly how far the conversion has progressed.

CONDITIONS FOR CONVERSIONS

The conditions that can occur during the conversion of values are described here. They are:

```
size
fixedoverflow
overflow
underflow
conversion
stringsize
```

A general discussion of conditions appears later, in Section XIII, "Condition Handling"; only the relevance of the conditions to the conversion of values is discussed here.

A program can establish an 'on' unit that is executed when a particular condition occurs. If an appropriate 'on' unit is not established when a given condition occurs, PL/I supplies a default 'on' unit. For some conditions, it is not valid for an 'on' unit to return control to the point at which the condition occurred; and such conditions usually abort execution of the program. For other conditions, a plausible recovery action is taken when execution resumes at the point at which the condition occurred.

'size' and 'fixedoverflow' Conditions

Two conditions can occur during the conversion of a value to a fixed-point arithmetic value; they are as follows:

- The 'size' condition occurs when a value is converted to a fixed-point value and the target precision does not specify enough digits to the left of the point to accommodate the magnitude of the given value.
- The 'fixedoverflow' condition sometimes occurs when the 'size' condition would otherwise occur.

These conditions have the same purpose, but they are implemented in different ways. If the 'size' condition is enabled, PL/I detects every case in which the magnitude of a converted value exceeds the capacity of the target precision. For example, an attempt to assign the value 12 to a 'fixed binary(3,0)' target is detected, even though the implementation may have allowed more than three bits for the value in hardware storage. The checking has a significant cost because it often must be performed by compiled instructions rather than hardware circuitry. In contrast, the 'fixedoverflow' condition occurs only when the hardware detects a value which cannot fit in a hardware register.

It is invalid for an 'on' unit to return control to the point at which either of these conditions occurred, and so there is no simple recovery method. PL/I provides a default 'on' unit that writes a message on the 'error_output' stream and then signals the 'error' condition. The effect of this default is to abort the execution of the program, and this is usually the appropriate response to a value that is too large.

'overflow' and 'underflow' Conditions

Two conditions are associated with the conversion of a value to a floating-point arithmetic value; they are as follows:

- The 'overflow' condition occurs when the conversion produces a floating-point value whose exponent is greater than 127.
- The 'underflow' condition occurs when the conversion produces a floating-point value whose exponent is less than -128.

Both of these conditions usually indicate a programming error; however, the PL/I processor treats the two conditions in different ways.

The 'overflow' condition is handled as a fatal error, similarly to the 'size' and 'fixedoverflow' conditions. An 'on' unit cannot return control to the point of occurrence, and the default 'on' unit writes a message and signals 'error', thus aborting program execution. On the other hand, PL/I does not treat the 'underflow' condition as a fatal error. Execution can resume at the point of occurrence, and in that case PL/I sets the value in question to zero. The default 'on' unit writes a message on the 'error_output' stream, but it then returns control to the point of occurrence with a zero value.

'conversion' Condition

When a conversion calls upon a character string to supply an arithmetic or bit-string value, the character string must have contents that allow such an interpretation. When this requirement is not met, the 'conversion' condition occurs. Some examples of character strings that cannot be converted to arithmetic targets follow, together with their corrected forms:

<u>Incorrect</u>	<u>Corrected</u>
"fifteen"	"15"
"+ 1 29"	"+29"
"20*3.14"	"62.8"
"-8.128422+02"	"-8.128422e+02"

The examples reflect the rules given earlier, in the definition of conversion for an arithmetic target.

An 'on' unit that is established for the 'conversion' condition can examine and modify the character string that caused the 'conversion' condition; and, after taking suitable remedial action, the 'on' unit can return to the point of occurrence. The facilities for this action are described in Section XIII, "Condition Handling." It is rarely possible to make a useful correction to a "bad" character string, and the PL/I default 'on' unit, which writes a message on the 'error_output' stream and signals the 'error' condition, is usually the appropriate action.

'stringsize' Condition

When a character-string or bit-string value is converted for a target whose length cannot accommodate the value, the 'stringsize' condition occurs. PL/I permits recovery from this condition. If the 'on' unit invoked by the condition returns to the point of occurrence, exactly enough characters or bits are truncated from the right end of the string value to reduce its length to that of the target. The default 'on' unit does not place a message on the 'error_output' stream; it returns directly to the point of occurrence with a truncated string value.

Guidelines for Conversion Conditions

The simplest policy with respect to the conditions that occur during value conversion is to treat them all as fatal errors. PL/I partially supports this policy by providing default 'on' units for most of the conditions that abort program execution. The two conditions that are not handled in this way are 'underflow' and 'stringsize', and their cases must be discussed separately.

Although PL/I does not abort program execution by default when an underflow occurs, the underflow message on the 'error_output' should be viewed as an error report. In most computations, an underflow is just as indicative of an error as an overflow; both occur when a computation has not been properly planned. When analysis of a computation shows that an underflow could occur, the underflow should be forestalled by programmed tests that detect the development of excessively small values.

Under certain circumstances, a more advanced approach to these conditions may be required. Suppose a system for the interactive performance of calculations is to be written as a PL/I program. In such a system, the user enters commands and these commands are interpreted by the PL/I program. Suppose the user gives a command whose interpretation causes an overflow to occur. The proper response is not to abort the execution of the whole interpretive program but rather to abort the command that the user entered. For this purpose, the PL/I program could establish a special 'on' unit for the 'overflow' condition.

SECTION V

PROGRAM SYNTAX

This manual gives an informal definition of the syntax of PL/I, and conveys much of this definition by means of examples. In contrast, the Multics PL/I Language Manual gives a complete and exact definition of the syntax, and expresses the definition in a special notation called BNF.

The syntax has two purposes. First, it is used to determine whether or not a given sequence of characters is syntactically valid. Some syntactically valid programs turn out to be invalid on other grounds, but the syntax does narrow the field in a reliable and effective way. Second, the syntax gives names to the components of a program. That is, the syntax defines precisely which character sequences are integers, which are expressions, and so on; therefore, these words take on an exact meaning in the discussion of a PL/I program.

This section does not give a detailed syntax for PL/I; instead, it presents the main syntactic features of PL/I. The details of syntax are given elsewhere in this manual; for example, the syntax of expression is given in Section VIII, "Expressions," and the syntax of the 'do' statement is given in Section XI, "Program Flow." A detailed syntax for each PL/I statement is also given, in reference form, in Appendix A, "A Guide to PL/I Statements."

This section presents four views of the syntax of a PL/I program. First, it considers a program to be a sequence of characters, and describes the set of allowed characters without discussing which sequences of characters are allowed. Second, it considers a program to be a sequence of lexemes, and defines the identifiers, constants, operators, and so on. Third, it considers a program to be a sequence of statements, and describes the general syntax of statements without entering into the details of individual statement syntax. Finally, it considers the program structures of PL/I that are used to gather sequences of statements together into single program components. The section concludes with a discussion of the relation between the external procedure and a program.

CHARACTERS

A program can be viewed simply as a sequence of characters. Any of the 128 characters of the ASCII character set can appear in a Multics PL/I program. A simple classification of those characters as they are used by Multics PL/I follows:

<u>letters:</u>	A B C ... Z a b c ... z
<u>digits:</u>	0 1 2 ... 9
<u>special characters:</u>	+ - * / = > < ^ & ! . , : ; () " _ \$ %
<u>spaces:</u>	<u>blank</u> <u>tab</u> <u>newline</u> <u>newpage</u> <u>vertical-tab</u>
<u>strings only:</u>	(the remaining ASCII characters)

The last item in the classification, "strings only", is included because every ASCII character, even if it has no other significance in Multics PL/I, can appear in a character-string constant.

The set of characters required for PL/I is relatively small; and all but four or five of the characters are found on a standard typewriter. Therefore, it is convenient to type and publish PL/I programs on conventional, noncomputerized equipment.

LEXEMES

A program can be viewed as a sequence of lexemes. The division of a program into lexemes corresponds to the division of ordinary text into words, punctuation marks, and spaces. As an example of the division of PL/I text into lexemes, consider the following assignment statement:

```
alpha= 5.66;
```

This statement is a sequence of five lexemes, as follows: the identifier 'alpha', the operator '=', the separator ' ' (blank), the literal constant '5.66', and the punctuator ';'. A long string of terminology can be applied to a single lexeme; for example, '5.66' is, in fact, a "fixed-point decimal real arithmetic literal-constant".

In what follows, each kind of lexeme is described; there are eight kinds of lexemes, as follows:

- identifier
- literal constant
- punctuator
- operator
- picture
- isub
- %include
- separator

After the lexemes are described, the separation rule for lexemes is given; it is the rule that governs blanks, newlines, and so on, to keep the lexemes of a program from running together. The discussion of lexemes concludes with a detailed classification of the lexemes.

Identifiers

An identifier is either a single letter or a letter followed by a sequence of characters; each character of the sequence must be a letter, a digit, an underscore, or a dollar sign. An identifier can be up to 256 characters long, so its length is, practically speaking, unlimited.

Often identifiers use only letters and digits. Examples of such identifiers are:

```
alpha ALPHA ALPha alpha23 alpha23xyz
```

Upper and lower case characters are distinct, so all of the identifiers just given are interpreted as different from one another.

An underscore character is used where, in English, a hyphen would be used. The hyphen is not available in PL/I because it cannot be distinguished from a minus sign. Examples of identifiers with underscore characters are:

```
account_name intake_manifold_pressure
```

Most Multics subroutine names end with an underscore character. Therefore, in order to avoid confusion and possible identifier conflict, a programmer should not introduce a name that ends with an underscore character.

A dollar sign should be used only in the name of a 'static external' variable, and it should be used in the manner described later, in Section VII, "Storage Management." An example of such an identifier is:

```
alpha$beta
```

This identifier refers to the variable 'beta' in the segment named 'alpha'.

KEYWORD VS. NAME

A particular occurrence of an identifier in a PL/I program is either a keyword or a name. When an identifier is used as a keyword, it has a specific meaning that is part of the definition of PL/I; for example, when the identifier 'goto' is used as a keyword, it always specifies a transfer of control. In contrast, when an identifier is used as a name, its meaning depends on its declaration in the program; for example, the identifier 'x' can be declared as a 'fixed dec(8)' variable, a 'label' constant, or any other kind of name.

In some programming languages, the identifiers that are used as keywords are reserved for that purpose only. However, in PL/I there are so many keywords that it would be a considerable disadvantage to reserve all of them, and so PL/I does not have this restriction. Instead, the interpretation of an identifier depends on its position in the syntax of a program. Suppose, for example, that a statement begins with the identifier 'goto'. The statement certainly could be a 'goto' statement; for example:

```
goto LAB;
```

In this case, the identifier 'goto' is interpreted as a keyword. However, a statement that begins with 'goto' could be an assignment statement; for example:

```
goto = 1;
```

In this case, the identifier is a name (and it must be properly declared). Thus the identifier 'goto' can be either a keyword or a name, depending on the context in which it appears.

It might be thought that there are cases in which it is difficult to determine whether an identifier is being used as a keyword or a name. This is not the case. In practice, an elementary knowledge of PL/I is sufficient to determine the interpretation of the identifiers in a program.

As an example of the interpretation of identifiers, consider the following program, in which keywords are underlined and names are not:

```
P:  proc;  
    dcl (sysin,sysprint) file;  
    dcl data float bin;  
    get data(data);  
    data = data**2;  
    put data(data);  
    end;
```

(The underlining in this example is only for the purposes of discussion; in a true PL/I program, an identifier is never underlined.) In this example, 'P', 'sysin', and 'sysprint' are used as names, 'data' is used both as a name and a keyword, and the other identifiers are used as keywords.

GUIDELINES FOR IDENTIFIERS

A programmer can use as names whatever identifiers are convenient. The choice of names does not affect the meaning of a program or its efficiency, but it does affect the readability of the program. The larger the program, the more important is the choice of name. The following suggestions apply:

- Where possible, use a long and descriptive name for a variable that is only referenced a few times. It is not much trouble to write out so few references, and the reader can understand the meaning of such a name immediately. On the other hand, use a short, abbreviated name for a variable that is referenced many times. In such a case, the resulting compactness is worth the trouble of introducing an abbreviation.
- When abbreviations are used, choose them according to some uniform rules, so that similar variables have similar names.

- Avoid using common keywords as names. When names such as

```
goto declare dcl if then
```

and so on are used as names, a superficial but irritating confusion is introduced. On the other hand, do use uncommon keywords as names where that is convenient. There is certainly no harm in using 'dft' to name a variable for the "debit final total" (or something of the sort) even though 'dft' is a keyword.
- Where possible, avoid using troublesome letters in identifiers. For example, the digits zero and one are troublesome because some output devices do not clearly distinguish between zero and the letter '0' or between one and the letter '1'.

Literal Constants

There is a literal constant lexeme for each type of arithmetic and string value. The full syntax and interpretation of these lexemes are given later, in Section VIII, "Expressions." The following is a representative set of examples of arithmetic literal constants:

<u>Arithmetic Constant</u>	<u>Data Type</u>
304	fixed dec(3)
3.04	fixed dec(3,2)
3.04e-5	float dec(3)
3.04e-5i	complex float dec(3)
0110001b	fixed bin(7)
011.0001b	fixed bin(7,4)
011.0001e-2b	float bin(7)
011.0001e-2bi	complex float(7)

Observe that an arithmetic constant does not begin with a sign. When a negative constant is required, it is written as two lexemes, a sign followed by an arithmetic constant.

The following is a representative set of examples of string literal constants:

<u>String Constant</u>	<u>Data Type</u>	<u>Remark</u>
"abcd"	char(4)	
(3)"abcd"	char(12)	means "abcdabcdabcd"
""	char(0)	means the null character-string
"" "Hello," "" he said."	char(17)	"" counts as " in value
"11101"b	bit(5)	
(4)"01"b	bit(8)	means "01010101"b
"b"	bit(0)	means the null bit-string

Any ASCII character can be used in a 'character' string constant, including such nonprinting characters as tab, newline, and so on. A string constant is a single lexeme, and is not considered to contain smaller lexemes.

Punctuators

There are six punctuator lexemes; each is given, together with its purpose, in the following table:

<u>Punctuator</u>	<u>Purpose</u>
. (period)	indicates the decimal or binary point; also, separates names in a qualified reference
, (comma)	separates items in a list of arguments, parameters, subscripts, declarations, options, and so on
: (colon)	terminates a condition prefix or a label prefix, separates the bounds of an array, and also appears in the range option of a 'default' statement.
; (semicolon)	terminates a statement
((left parenthesis)	indicates the beginning of a list, an expression, an iteration factor, and so on
) (right parenthesis)	indicates the end of a list, an expression, an iteration factor, and so on

These lexemes are used in most of the features of PL/I.

Operators

There are five kinds of operator lexemes; they are defined as follows:

<u>Classification</u>	<u>Operators</u>
arithmetic	+ - * / **
relational	= ^= < ^< > ^> <= >=
logical	^ &
string	
qualifier	->

Most of the operators are defined in Section IX, "Operations." The only exception is the qualifier operator, which is defined in Section VIII, "Expressions."

Pictures

The picture lexeme is a specialized lexeme that is used to specify the format of a character string. It begins and ends with a quote, but the characters that appear between quotes are restricted. An example of a picture is:

```
"$9,999.99"
```

This picture specifies a character-string of length nine that consists of a dollar sign, followed by a digit, followed by a comma, followed by three digits, followed by a period, followed by two digits. In other words, it describes a six-digit, dollars-and-cents figure.

A picture lexeme is used in only two contexts. It is used in a 'picture' attribute in the declaration of a pictured variable; this usage is described earlier, in Section III, "Value Storage." Second, it is used in a picture format item in an edit-directed input/output statement; this usage is described later, in Section XIV, "Stream Input/Output." In both cases, the picture itself is interpreted in the same way.

Isubs

The isub lexeme is a very specialized lexeme that is used only in the 'defined' attribute. It is composed of an unsigned integer followed by the three letters 'sub'; for example, '5sub'. Consider the following declaration of the array 'B':

```
declare B(2,4) defined A(2sub,3*1sub);
```

The first isub lexeme is '2sub' and means, "the value of the second subscript in a reference to 'B'". The second isub lexeme is '1sub' and means, "the value of the first subscript in a reference to 'B'". Thus, for example, the statement:

```
B(i-1,j) = 1;
```

is interpreted as:

```
A(j,3*(i-1)) = 1;
```

'%include' Macro

A '%include' macro is a sequence of three lexemes, and has the following form:

```
%include psn ;
```

where psn is the partial segment name. The partial segment name can be either an identifier or a character-string constant. The '%include' macro is discussed here, under "Lexemes", because it begins with a lexeme that has a special form (it begins with a percent character) and because its effect is to modify the lexical content of a program.

MACRO INTERPRETATION

A '%include' macro directs the compiler to obtain an include file; it is the only construct in the language that is not merely translated by the compiler for later execution. In response to an '%include' macro, the compiler takes the following steps:

1. Form a segment name by dropping the quotes from the partial segment name (if it is a character-string constant) and adding the suffix '.incl.pl1'. Locate the Multics segment that is designated by this name.
2. Compile the given program as if the '%include' macro were replaced by a blank, followed by the contents of the segment located in Step 1, followed by a blank.

This definition of the '%include' macro requires further explanation, as follows:

- When the compiler obeys a '%include' macro, it does not modify any of the programmer's files. It behaves as if the given program were modified.
- The contents of an included segment can, itself, use a '%include' macro, and that macro is interpreted just as if it appeared in the given program.
- A character-string constant is permitted for use as a partial segment name because Multics segment names are not always identifiers. Thus,

```
    %include "alpha.test";
```

can be used to designate the contents of the segment designated by 'alpha.test.incl.pl1'.
- In Step 2, a blank is inserted before and after the included text to keep the first and last lexeme from running into neighboring lexemes.

In order for the language translator to find include files, the set_search_paths command is used with the pl1 search list. This command description is found in the MPM Commands.

MACRO EXAMPLE

As an example of the expansion of a '%include' macro, suppose that the Multics segment named 'test.pl1' exists and contains a complete procedure, as follows:

```
P:  proc;
    dcl alpha float bin;
    dcl beta(20) char(10);
    %include xpool;
    ...
end;
```

Each time this segment is compiled, the compiler finds the segment 'xpool.incl.pl1' and replaces the '%include' macro by the contents of that segment. Suppose 'xpool.incl.pl1' exists and contains the following declarations:

```
dcl 01 x external static,
      02 size fixed bin,
      02 tab(1000) char(10);
dcl cnt fixed bin;
```

For the particular compilation under consideration, the affect is as if 'test.pl1' contained the following material:

```
P:  proc;
      dcl alpha float bin;
      dcl beta(20) char(10);
      dcl 01 x external static,
            02 size fixed bin,
            02 tab(1000) char(10);
      dcl cnt fixed bin;
      ...
      end;
```

The actual contents of 'test.pl1' are not changed as the result of the compilation.

GUIDELINES FOR MACROS

The '%include' macro is often used to assist in the organization of a large program. Such a program is usually divided into several external procedures, which are written and compiled separately and then brought together for execution. The external procedures usually have certain text in common, such as the declaration of variables and routines that are used throughout the program. This common text can be handled as follows:

- Create a common segment, which is a segment whose name ends with 'incl.pl1' and whose contents is the common text.
- In each external procedure, write a '%include' macro that references the common segment.

The important advantage of this approach is that just one copy of the common text exists, and therefore changes are made in just one place.

This use of common segments can extend through several levels of program organization. Suppose that a program is divided into subprograms and each subprogram is a set of sub-subprograms. Each sub-subprogram can use a '%include' macro to reference a common segment for the subprogram. That common segment can, in turn, use a '%include' macro to reference a common segment for the entire program.

The application for the '%include' macro that has just been described is an important one; however, it is not the only possible use. The segment designated by a '%include' macro need not be a sequence of statements; it can be any sequence of lexemes that make sense in the context in which they are used.

Separators

There are two kinds of separator lexemes: the space and the comment. They are defined as follows:

- A space lexeme is one of the following characters:

blank
tab
newline
newpage
vertical-tab

Although these characters have different effects, each of them usually has the effect of leaving empty space in the listing of a program; that is why they are called "spaces".

- A comment lexeme has the following form:

```
/*cs*/
```

where cs is the comment string. The comment string is any sequence of ASCII characters that does not contain '*/'. A comment is used to insert a message that is directed to the human reader of a program but that is ignored by the PL/I processor.

Separator lexemes are used interchangeably. That is, any sequence of one or more separator lexemes is equivalent to any other sequence of one or more separator lexemes. The role of separator lexemes in PL/I is given by the separation rules, which are described in the following paragraphs.

Separation Rules

The subsequent sections of this manual give many informal rules for the syntax of PL/I. Each rule ultimately specifies that a given construct is a certain sequence of lexemes. However, the definitions make no mention of separators. Instead, they assume that the following separation rules are followed in all cases:

1. One or more separators can appear between any sequence of two lexemes. This rule permits the use of spaces to control layout and the addition of comments to provide explanations; thus a program can be made intelligible to a human reader.
2. One or more separators must appear between any sequence of two lexemes if the first is an identifier, literal constant, or isub and the second is also an identifier, literal constant, or isub. This rule requires the use of a separator where two lexemes might otherwise run together to form a single lexeme.
3. A separator must not appear within a lexeme. This rule prevents a lexeme from being broken up into two lexemes.

Consider an example of the application of these rules. One version of the 'allocate' statement is defined (in Section VII, "Storage Management") as having the following form:

```
allocate id ;
```

where id is an identifier. This definition is satisfied by each of the following four constructs:

```
allocate beta_5;
```

```
allocate    beta_5    ;
```

```
allocate  
    beta_5;
```

```
allocate beta_5 /*the stress variable*/;
```

All of these statements mean exactly the same thing to the PL/I processor, but the first separation rule has been applied to produce differences that are important to a human reader.

A violation of the separation rules is quite obvious to a human reader, and that is why those rules can be given once here and then assumed to apply throughout the rest of the manual. Consider the following construct:

```
allocatebeta_5;
```

This construct is not a valid PL/I statement. It arose from the definition of the 'allocate' statement, but the second separation rule was violated, allowing 'allocate' and 'beta_5' to run together and form a single identifier.

As a second example of the violation of the separation rules, consider the following construct:

```
all ocate beta_5;
```

This construct is not a valid PL/I statement. Once again, it arose from the definition of the 'allocate' statement, but the third separation rule was violated, causing the keyword 'allocate' to be broken up into two identifiers.

Classification of Lexemes

The following list gives a complete classification for the lexemes of PL/I. It shows, for example, that there are two kinds of decimal real arithmetic literal-constant lexemes; namely, fixed-point and floating-point. The complete set of punctuators and operators and representative examples of other lexemes are given at the right.

lexeme

<u>identifier:</u>	x declare rate_of_change table\$sum2
<u>literal constant</u>	
<u>arithmetic</u>	
<u>real</u>	
<u>decimal</u>	
<u>fixed-point:</u>	59 18. 6.8923 .0030
<u>floating-point:</u>	8.23e+3 1e-5
<u>binary</u>	
<u>fixed-point:</u>	1101b 101.b 10011.101b .0001b
<u>floating-point:</u>	101.0001e+2b 1111e-5b
<u>imaginary</u>	
<u>decimal</u>	
<u>fixed-point:</u>	59i 18.i 6.8923i .0030i
<u>floating-point:</u>	8.23e+3i 1e-5i
<u>binary</u>	
<u>fixed-point:</u>	1101bi 101.bi 10011.101bi .0001bi
<u>floating-point:</u>	101.0001e+2bi 1111e-5bi
<u>string</u>	
<u>character-string:</u>	"abc" "Say ""No""." (5)"x" ""
<u>bit-string:</u>	"110"b (36)"1"b ""b
<u>punctuators:</u>	. , : ; ()
<u>operators</u>	
<u>arithmetic:</u>	+ - * / **
<u>relational:</u>	= ^= < ^< > ^> <= >=
<u>logical:</u>	^ &
<u>string:</u>	
<u>qualifier:</u>	->
<u>picture:</u>	"s999v.99" "\$\$\$\$9v.99cr"
<u>isub;</u>	1sub 5sub
<u>%include:</u>	%include Table; %include "rd>x>sm";
<u>separator</u>	
<u>space:</u>	(blank, tab, newline, newpage, and vertical-tab characters)
<u>comment:</u>	/* Ignore this message. */

STATEMENTS

A program can be viewed as a sequence of statements. The notion of the statement is one of the important features of the high-level languages. A PL/I programmer thinks of the statement as the executable unit of programming just as an assembly language programmer thinks of the hardware instruction as the executable unit of assembly language programming. Yet a single PL/I statement generally corresponds to five or six hardware instructions -- sometimes less, often more. Therefore, from a programmer's point of view, the use of PL/I instead of assembly language reduces the number of executable units in a program by a large factor.

Statement Prefix

The following syntax rules apply to all PL/I statements:

- A statement is a prefix, followed by a statement body, followed by a semicolon.
- The prefix of a statement consists of an optional sequence of condition prefixes followed by an optional sequence of label prefixes.
- A condition prefix is a parenthesized list of condition prefix names separated by commas and followed by a colon.
- A label prefix is an identifier followed by a colon.

CONDITION PREFIXES

The purpose of a condition prefix is to alter the set of conditions that are enabled during the execution of a statement or a block. When a condition is enabled, the corresponding error checking is performed with the resulting enhancement of the debugging process. When a condition is disabled, checking for the corresponding error condition is not usually performed and the program executes at less cost. Consider the statement:

```
(subscriptrange, nosize): z(i) = alpha;
```

The 'subscriptrange' condition prefix name enables the corresponding condition and, in effect, forces the processor to check to see if the value of 'i' is outside the range of the bounds of the array 'z'; that policy is good for debugging and bad for optimization. On the other hand, the 'nosize' condition prefix name disables the size condition and frees the processor from the responsibility for detecting a value of 'alpha' whose magnitude is too large for an element of the array 'z'; that policy is bad for debugging but good for optimization. Further details are given in Section XIII, "Condition Handling."

LABEL PREFIXES

The purpose of a label prefix is to declare a program address constant name and to specify its value. Three cases apply:

- If a label prefix is in a 'procedure' or 'entry' statement, then the identifier is declared as an 'entry constant'.

- If the label prefix is in a 'format' statement, then the identifier is declared as a 'format constant'.
- If the label prefix is in the prefix of any statement not mentioned in the preceding cases, then the identifier is declared as a 'label constant'.

The classification just given indicates that, although the terminology "label prefix" is convenient, the identifier in a label prefix is not necessarily a label constant. Further details are given in Section XI, "Program Flow."

Statement Body

There are about 25 different kinds of statements in Multics PL/I; but the statements make extensive use of a small repertoire of components and obey some simple rules. The observations that follow are not recommended for study or memorization; instead, they are intended to assist the reader in recognizing the patterns that make the syntax of PL/I relatively easy to read.

- Keywords. Except for the assignment statement and the null statement, every statement body begins with a keyword. The complete syntax for each statement is given in Appendix A, and the entries in that appendix are arranged in alphabetical order according to the initial keyword.
- Options. The main part of most kinds of statement is a sequence of options. An option is a keyword followed by something in parentheses. An option specifies a subcommand within a statement, and can often be omitted from a statement without rendering the statement invalid (hence the name "option"). For example, consider the statement:

```
put file(beta) skip(2) list(x,y);
```

This statement consists of a keyword 'put' and a sequence of three options. When the first two options are omitted, the statement is:

```
put list(x,y);
```

In this statement, the missing options are treated in two different ways. In the absence of the 'file' option, PL/I assumes 'file(sysprint)'; but in the absence of the 'skip' attribute, PL/I does not perform the 'skip' action.

- Clauses. In a rather small number of cases, a clause is used instead of an option. A clause is an option without the parentheses. In a 'do' statement, each clause is a keyword followed by an expression. For example, consider:

```
do i = 1 to 20 by 2 while(u = v);
```

In this statement, 'to 20' and 'by 2' are clauses, but 'while(u=v)' is an option. (A common error is to write the 'while' option as a clause, without the parentheses.)

- Attributes. An attribute is a keyword that is sometimes followed by something in parentheses. It thus resembles an option; but it is used to give the declaration of an identifier rather than to give a subcommand. Attributes appear in 'declare' statements and, in a restricted way, in several other kinds of statements. For example,

```
declare x real fixed binary precision(8,2);
```

This statement contains four attributes, the last of which has a parenthesized list of two integers.

- Spaces. In a sequence of options that describe a single action (such as the opening of one file or the allocation of one storage unit), the options are separated by spaces rather than by commas. A similar rule applies to the attributes that describe a single identifier.
- Commas. Several statements use commas to express in one statement what would otherwise require several statements; for example,

```
declare x decimal fixed(8), y float;
```

is equivalent to

```
declare x decimal fixed(8);  
declare y float;
```

The punctuation used in such compound statements is always the comma (never the space), and the comma can be thought of as a "little semicolon" when it is used in this way. Although the use of one statement for several does save some writing effort, it does not change the cost of compiling or executing the statement; and there are some disadvantages to the use of a compound statement. A compound statement makes examination or editing of a program more difficult, and since Multics compiler diagnostics are keyed to complete statements (not phrases within statements), a diagnostic for a complicated statement may be ambiguous.

Classification of Statements

A classification of all the Multics PL/I statements is given here. The classification is according to the principal function of each statement. There is a section in this manual for each of the eight kinds of statement given in the classification.

statement

declaration

declare dcl x fixed dec(5,2) static initial(1);
default dft (variable & range(c)) character(1);

storage management

allocate alloc alpha in(tab) set(ip);
free free Q;

assignment

(no keyword) x(i-3) = y*(sin(theta)-omega);

program flow

begin begin;
end end;
do do k = 1 to n+5 while(m<0);
goto goto LAB;
if if beta = -2 then q = r-s; else q = 0;
(null statement) ;

procedure invocation

procedure P: proc(x,y) returns(float);
entry Q: entry(a1,a2);
end end;
call call TCC(a-3*phi,H3);
return return(r-2*z);
stop stop;

condition handling

on on endfile(sysin) goto EXIT2;
revert revert endpage(report);
signal signal error;

stream input/output

open open file(AWC) print linesize(80);
close close file(test2);
get get file(ww) edit(x,y)(p"zzz.99bbbb");
put put file(drop) page;
format F: format(a(10),p"bbb--9v.99");

record input/output

open open file(a) keyed sequential update;
close close file(insp);
read read file(cust) into(tail);
write write file(rec) keyfrom(x) from(y);
delete delete file(employee) key(ssno);
rewrite rewrite file(m) key(a3) from(beta);
locate locate buf set(p3) file(arc);

A single example of each statement is given at the right. The selection of the example is necessarily arbitrary. Every example has a null prefix except for those statements that require at least one label prefix.

PROGRAM STRUCTURES

There are three PL/I constructs that group a sequence of statements into a single program structure: the group, the 'procedure' block, and the 'begin' block. Each of these structures can be compared to the paragraph of conventional text; however, they are more powerful than the conventional paragraph. The additional power comes from the fact that any one of these program structures can, itself, be regarded as a single statement. Thus a group or block can be included in the sequence of statements encompassed in a larger group or block. This arrangement of one program structure within another is called nesting.

Groups

There is just one kind of group, the 'do' group. A 'do' group has the following form:

- A 'do' statement followed by
- A sequence of constructs, each of which is a statement, a group, or a block, followed by
- An 'end' statement.

Groups and blocks cannot overlap but must be nested; that is, any 'do' statement, 'procedure' statement, 'begin' statement, 'entry' statement, or 'end' statement that is contained in a given 'do' group must be part of a complete group or block that is contained in the given 'do' group.

The simplest application of a 'do' group does nothing more than gather a sequence of statements into a single executable unit. Consider, for example, the following 'if' statement:

```
if x = 0
  then do;
    y = 1;
    z = 2;
  end;
```

This statement sets the variables 'y' and 'z' only if 'x' is zero. The 'do' group is treated as a single construct governed by the 'if' statement. If the 'do' and 'end' statements are omitted, then the example becomes:

```
if x = 0
  then y = 1;
    z = 2;
```

Now the meaning is different; the example consists of two statements, and the assignment to 'z' occurs regardless of whether 'x' is zero or not. Therefore, the layout of the revised example is misleading and it should be written as follows:

```
if x = 0
  then y = 1;
z = 2;
```

This layout makes it clear that only the assignment to 'y' is governed by the 'if' statement.

In the more complicated applications of a 'do' group, it not only gathers a sequence of statements into a single executable unit, but also executes the statements repeatedly. Consider, for example, the following group:

```
do i = 1 to 20;
  a(i) = b(n+1-i);
end;
```

This statement performs the assignment statement 20 times. For each execution of the group, the index variable, 'i', assumes the values '1', '2', and so on up to '20'. There are many variations in the repeating group, and these are described later, in Section XI, "Program Flow."

Blocks

There are two kinds of blocks: the 'procedure' block and the 'begin' block. They have the following form:

- A 'procedure' statement or 'begin' statement, depending on whether the block is a 'procedure' or 'begin' block, followed by
- A sequence of constructs, each of which is a statement, a group, or a block, followed by
- An 'end' statement.

An 'entry' statement can appear in a 'procedure' block. Aside from this, any 'do' statement, 'procedure' statement, 'begin' statement, 'entry' statement, or 'end' statement that is contained in a given block must be part of a complete group or block that is contained in the given block; that is, groups and blocks must be nested.

The two kinds of block both define a scope for the declaration of identifiers. Within a given scope, an identifier can have a meaning that is entirely different from the meaning of the same identifier outside of the given scope. Thus a programmer can write a procedure that is intended for use within a large program without concern for conflict between his use of identifiers and that of other programmers. In addition, he can indicate by his use of scopes the distinction between a variable that is used just in a small procedure and one that is used throughout the procedure he is writing. The importance of scopes is so great that 'procedure' blocks and 'begin' blocks are discussed together here, even though they are quite different in the way they are executed.

A 'procedure' block is executed as a closed subroutine; that is, it is executed by means of a 'call' statement or a function reference, and not when flow reaches it from the preceding statement. In contrast, a 'begin' block is executed in-line; that is, it is executed when flow reaches it from the preceding statement or by transfer of control to its 'begin' statement. (The 'begin' block in an 'on' statement is handled in a special way.)

Of the two kinds of block, the 'procedure' block is by far the more important. A PL/I program is a collection of one or more procedures, and every subroutine within a program is written as a 'procedure' block. Consider the following example:

```

P:    proc;
      dcl (x,y,z) float bin;
      dcl (sysin, sysprint) file;
      get list(x,y);
      call DIST(x,y,z);
      put list(z);
DIST: proc(c1,c2,r);
      dcl (c1,c2,r) float bin;
      dcl sqrt builtin;
      r = sqrt(c1**2+c2**2);
      end;
      end;

```

This example is a 'procedure' block and it is a complete program. It contains a smaller 'procedure' block that is a subroutine, and that is invoked by the 'call' statement. A complete discussion of 'procedure' blocks is given later, under "Procedure Invocation".

A 'begin' block is rarely used. It is sometimes essential in an 'on' statement, as described later under "Condition Handling". Usually, however, a need to gather statements together can be handled better by a 'do' group or a 'procedure' block. It is not unusual for a large program to be written without the use of any 'begin' blocks.

Summary of the Program Structures

The following table shows various properties of the three kinds of program structure:

	<u>Group</u>	<u>'begin' Block</u>	<u>Procedure</u>
Gathers statements into a single executable unit.	yes	yes	yes
Can iterate the gathered statements.	yes	no	no
Is executed/skipped when reached by sequential flow of control.	executed	executed	skipped
Can be called as a closed subroutine.	no	no	yes
Defines a <u>scope</u> for the declaration of identifiers.	no	yes	yes

EXTERNAL PROCEDURES AND THE PROGRAM

Throughout most of this manual, the terms "external procedure" and "program" are used interchangeably. No harm comes from that, because a single external procedure can be executed as a program. In the following paragraphs, however, the distinction between the two terms is described in detail.

An external procedure is a procedure that is not contained in any larger PL/I construct. It is called "external" precisely because it is not contained inside any other PL/I construct. An external procedure can contain other procedures; and these latter procedures are all internal procedures.

A program is a set of one or more external procedures that are executed in concert. Each external procedure is written, edited, filed, and compiled separately. It is possible to bind the external procedures into a single Multics object segment, but this binding is not required; it is only necessary that the object segments for the external procedures be available when the program is executed.

Any program can be written as a single external procedure; and in that case, the distinction between a program and an external procedure is not very important. But there are good reasons to divide a program of medium or large size into several external procedures. The reasons are:

- In a large project involving more than one programmer, it is important to be able to compile and test parts of a program separately.
- For programs of considerable size, compilation in parts is less expensive than compilation of the whole; this is simply a fact of compiler technology. Furthermore, when a program becomes very large, it exceeds the capacity of the compiler and cannot be compiled as a whole.
- To change a single statement of an external procedure the entire external procedure must be recompiled. The smaller the external procedures, the smaller the cost of making an isolated change.

The division of a program into external procedures can be carried to an undesirable extreme. When the division results in variables being shared between the two external procedures, these variables must be 'external' variables or procedure parameters; and the implementation of such variables is relatively expensive. In the absence of other guidance, external procedures should be kept between 100 and 1000 lines in length.

SECTION VI

DECLARATIONS

The use of a name in a program is a name reference and each name reference has a declaration. The declaration provides two items of information. First, it gives a set of attributes and (in the case of a structure variable) level numbers. Second, it associates the name reference with a particular block. Later sections of this manual describe how declarations are used in the interpretation of the different kinds of name references. This section describes the mechanism that supplies the declarations.

Each occurrence of a name in a program either supplies a declaration or uses a declaration. Consider the following fragment of a program:

```
    decl x float bin;
    ...
    x = 5;
```

In this example, the first occurrence of 'x' supplies a declaration, and the second occurrence of 'x' makes use of that declaration. If the declaration of 'x' were not supplied in this way, then the assignment statement could not be interpreted.

With one exception, the declaration of a name reference is determined by the PL/I compiler, once and for all, before a program is executed. The exception is for a variable extent: a variable array bound, a variable maximum string length, or a variable area size. To determine the declarations of the name references in a program, the compiler makes repeated use of two operations: the establishment of a declaration and the resolution of a name reference.

This section has three main parts. First, the constructs used for the establishment of declarations are described. Next, the rules for the resolution of names are presented. Finally, diagrams that show all possible declarations of names are provided and a complete classification of the attributes is given.

ESTABLISHMENT OF DECLARATIONS

This discussion begins with an example; then it describes the program constructs that are used in establishing declarations.

Example of the Establishment of Declarations

The following program contains several examples of the establishment of declarations:

```
P:  proc;
    decl x float bin;
    decl (sysin,sysprint) file;
L2:  . get list(x);
    call Q;
    if x = 0 then goto L2; else return;
Q:  proc;
    decl L2 float bin;
    L2 = x**2 - 3*x**2;
    put skip list(x,L2);
    end;
end;
```

The program has two blocks, which can be conveniently be called the "outer block" and the "inner block". The declarations explicitly established in the program are as follows:

- In the outer block:
 - 'x' with the attribute 'float bin'
 - 'sysin' with the attribute 'file'
 - 'sysprint' with the attribute 'file'
 - 'L2' with the attribute 'label internal constant'
 - 'Q' with the attributes 'entry internal constant'
- In the inner block:
 - 'L2' with the attribute 'float bin'
- In an imaginary 'begin' block that encloses the entire program:
 - 'P' with the attributes 'entry external constant'

More is said of the "imaginary" block later in this section.

Observe that two declarations are established for the identifier 'L2' and that the identifier is used in two different ways. Such a double use is not attractive when the uses are so close together; but in a large program, the repeated use of a single identifier is difficult to avoid. When the subject of resolution is discussed later in this section, the way in which PL/I decides which declaration applies to a particular use of 'L2' is given.

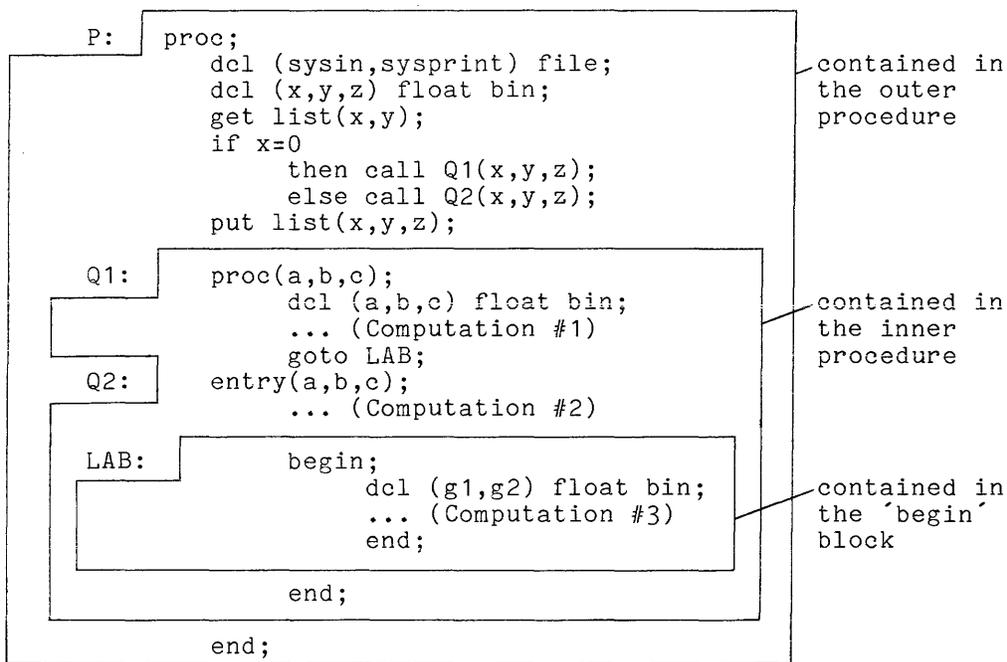
Containment and Immediate Containment

The establishment of declarations depends on the definitions of containment and immediate containment. These definitions follow:

- An occurrence of a program construct (such as a statement, a label prefix, or a name) is contained in a block if it is part of the 'procedure' or 'begin' statement with which the block begins, part of the 'end' statement with which the block ends, or part of any statement in between. However, there are two exceptions. First, any label prefixes in the 'procedure' or 'begin' statement that begins a given block are not contained in the given block. Second, any label prefixes in an 'entry' statement that is part of a given block but not part of a smaller block are not contained in the given block.
- An occurrence of a program construct is immediately contained in a given block if it is contained in the given block but not in any smaller block.

In the establishment of declarations, these definitions are used to determine which block immediately contains a given 'declare' statement or label prefix.

As an example of the application of these definitions, consider the following program:



This program is composed of three blocks; they can be referred to as the "outer procedure", the "inner procedure", and the "begin' block". The three outlines show which portions of the program are contained in each of the three blocks.

The following observations are typical of those necessary for the establishment of declarations:

- The label prefix 'P:' is not contained in any of the three blocks, but it is contained in an imaginary 'begin' block that is considered to enclose the entire program.
- The first two 'declare' statements are immediately contained in the outer procedure.
- The label prefixes 'Q1:' and 'Q2:' are immediately contained in the outer procedure (and are not contained in the inner procedure); this is true even though 'Q2:' occurs in the midst of the inner procedure.
- The third 'declare' statement and the label prefix 'LAB:' are contained in both the outer and inner procedures, but they are immediately contained in the inner procedure only.
- The fourth 'declare' statement is contained in all three blocks; but it is immediately contained in the 'begin' block only.

The imaginary 'begin' block mentioned in connection with the declaration of 'P' is a definitional artifice, called the root block; that makes the rules for the establishment of declarations simpler. The external procedures of a program are thought of as being enclosed in an imaginary 'begin' block. All of the entry constant names for the external procedures are declared in this 'begin' block.

'declare' Statement

The principal means for establishing a declaration is the 'declare' statement. As the keyword 'declare' suggests, the statement is devoted to supplying information. When a 'declare' statement is encountered in the course of program execution, it produces no action.

A 'declare' statement gives one or more declarations. Each declaration associates a set of attributes with a name. The declarations are established in the block that immediately contains the 'declare' statement.

The 'declare' statement is available in several forms. In many cases, a 'declare' statement is a simple declaration of a scalar or array variable name. When a structure variable is declared, a special form of the 'declare' statement is required. In order to reduce the amount of writing required, declarations can be combined and factored. These forms of the 'declare' statement are discussed in the following paragraphs.

SIMPLE DECLARATIONS

The simplest form of a 'declare' statement is the keyword 'declare', a name, a sequence of attributes, and a semicolon. Two examples are:

```
declare alpha_p5 decimal float bin static;
declare x;
```

The effect of the first statement is to establish in the immediately containing block a declaration of 'alpha_p5' with attributes 'decimal float static'. The second statement establishes in the immediately containing block a declaration of 'x' with no attributes. In both cases, the attributes not given in the 'declare' statement are filled in by PL/I according to default rules.

The following statement declares an array variable name:

```
declare beta dimension(3*n-1) float bin controlled;
```

When the 'dimension' attribute immediately follows the declared identifier, the keyword 'dimension' can be omitted. Thus this statement is usually written as

```
declare beta(3*n+1) float bin controlled;
```

The effect of this statement is to establish in the immediately containing block a declaration of 'beta' with the attributes 'dimension(3*n+1) float controlled'. The expression '3*n+1' is not evaluated when the declaration is established (before program execution); instead it is evaluated when storage for the array is allocated (during program execution).

STRUCTURE DECLARATIONS

Most names can be declared by means of the simple form of declaration just described. The only exception is the declaration of a structure variable name. The declaration of a structure requires the declaration of several names: one for the entire structure, others for its members, yet others for the members of each of its members, and so on. Each name is declared in a declaration clause, which consists of a level number, the name itself, and a sequence of attributes. All of the names for a given structure must be in the same 'declare' statement, and the declaration clauses are separated by commas.

Consider, for example, the following declaration of the structure variable 'subscriber':

```
declare 01 subscriber external,
        02 name,
          03 first char(15) varying,
          03 initial_of_middle char(1),
          03 last char(25) varying,
        02 serial_number decimal(9);
```

This statement contains six declaration clauses. It establishes the following declarations in the immediately containing block:

- 'subscriber', a structure with the attributes 'external' and with members 'name' and 'serial_number'
- 'name', a structure with no attributes and with members 'first', 'initial_of_middle', and 'last'
- 'first', a scalar with attributes 'char(15) var'
- 'initial_of_middle', a scalar with attribute 'char(1)'
- 'last', a scalar with attributes 'char(25) var'
- 'serial_number', a scalar with attributes 'decimal(9)'

SHORT FORMS OF DECLARATIONS

A major portion of a PL/I program is devoted to 'declare' statements, so PL/I has several features designed to shorten declare statements. The sole purpose of these features is to reduce the size of a program and thus make it easier to write and to read. The features do not have any effect on the cost of executing the program.

Abbreviations and Defaults

A useful feature is the abbreviation of the keyword 'declare' to 'dcl'. A second feature is the provision of many abbreviations and defaults for attributes. The designers of PL/I selected the defaults to cover the most commonly required attributes. It follows that when an identifier is used in some ordinary way not many attributes need be written explicitly. For example, in the 'declare' statement

```
dcl Q72 float bin static;
```

the programmer has omitted the defaults 'real binary precision(27) aligned internal variable' because these are supplied by default. The defaults for the storage type attributes are given in Section III, "Value Storage." Other default rules are given in Section VII, "Storage Management."

Combining Declarations

A sequence of 'declare' statements can be combined into a single 'declare' statement. Consider the statements:

```
dcl x float bin;  
dcl alpha fixed dec(10);  
dcl Q3 float bin;
```

A single, equivalent statement is:

```
dcl x float, alpha fixed dec(10), Q3 float;
```

Factoring Declarations

Common attributes can be factored from individual declarations in a combined 'declare' statement. Another equivalent form for the 'declare' statement above is:

```
dcl (x,Q3) float bin, alpha fixed dec(10);
```

This feature is called factoring because it is similar to the mathematical operation of factoring out a common multiplier from a sum.

Factoring can be applied more generally than the preceding example indicates. Factoring can be applied repeatedly to the same statement. For example, the statement:

```
dcl x float static, y float static, z float;
```

can be written equivalently as:

```
dcl (x static, y static, z) float;
```

and then as:

```
dcl ((x, y) static, z) float;
```

The result has two attributes in it instead of the original five.

Factoring can also be applied to the level numbers used in the declaration of a structure name; the only difference is that level numbers are factored to the left while attributes are factored, as before, to the right. For example, the statement:

```
dcl 01 position,  
    02 x float,  
    02 y float,  
    02 z float;
```

can be written as:

```
dcl 01 position,  
    02 (x, y, z) float;
```

GUIDELINES FOR 'declare' STATEMENTS

A 'declare' statement can be placed anywhere in the block in which its declaration is to be established, provided it is immediately contained in that block. It is suggested, however, that all 'declare' statements be placed immediately after the 'procedure' or 'begin' statement that begins the block. If this convention is followed, a human reader always knows where to look for the 'declare' statements and the 'declare' statements do not clutter up the portion of the block which is devoted to executable statements.

The extensive use of the combination and factoring of 'declare' statements can make a program difficult to read, debug, and edit. Furthermore, the Multics PL/I compiler keys its diagnostics to statements, not to individual declarations; therefore, a diagnostic message about a 'declare' statement with several declarations can be ambiguous.

The use of the outline layout of the declaration of structures, as shown in the examples in this section, is recommended. The leading zero in each level number is not required in PL/I; it is a stylistic device carried over from COBOL by some PL/I programmers. It has the advantage that it distinguishes the level numbers, which do not partake in the computational activity of PL/I, from the arithmetic constants.

Label Prefixes

A second means for establishing a declaration is the label prefix. A label prefix is an identifier followed by a colon and it always occurs immediately before a statement body or another label prefix. It is useful to have a common term, "label prefix" for all uses of this construct; but, in fact, the identifier in a label prefix can be a label constant name, an entry constant name or a format constant name.

LABEL CONSTANT NAMES

When a label prefix occurs before any statement except a 'procedure', 'entry', or 'format' statement, the identifier in the prefix is a label constant name. An effect of the label prefix is to establish in the immediately containing block a declaration of the identifier as 'label internal constant'. Another, more fundamental effect, is to label the statement so that it can be the destination of a transfer of control; that effect is described later, in Section XI, "Program Flow."

A parenthesized, optionally-signed integer can be used in a label prefix as a subscript to a label constant name. A given identifier can appear in several label prefixes in a single block provided each appearance has a different subscript. The effect of the set of label prefixes with the given identifier is to establish a single declaration of the identifier as 'label internal constant dimension(i1:i2)', where i1 is the smallest integer used as a subscript and i2 is the largest.

As an example of the use of label prefixes to declare label constant names, consider the following procedure:

```
MES:  proc(i);
      dcl sysprint file;
      dcl i fixed bin;
      goto LAB(i);
LAB(14): put list("x is too big"); goto EXIT;
LAB(-3): put list("z3 is negative"); goto EXIT;
LAB(1):  put list("a exceeds x"); goto EXIT;
EXIT:   end;
```

The label declarations established in this procedure are:

- 'LAB', with attributes 'label internal constant dimension(-3,14)'
- 'EXIT', with attributes 'label internal constant'

ENTRY CONSTANT NAMES

When a label prefix occurs before a 'procedure' statement or an 'entry' statement, the identifier in the prefix is an entry constant name. In the event that the procedure has neither arguments nor a result, the effect of the label prefix is to establish in the immediately containing block a declaration of an identifier with the attributes 'entry internal constant' (if the immediately containing block is not the root block) or 'entry external constant' (if the immediately containing block is the root block). When the procedure has arguments or a result, the attributes of the arguments or the result is included in the declaration of the entry constant name.

As an example of the use of label prefixes to declare entry constant names, consider the following procedure, which is assumed to be contained in some larger block:

```
REP:  proc(s,cnt) returns(char(1000) varying);
      dcl s char(1000) varying;
      dcl cnt fixed bin;
      dcl i fixed bin;
      i = cnt;
      goto L1;
REP2: entry(s) returns(char(1000) varying);
      i = 1;
L1:   ...
      end;
```

The first two label prefixes establish the following declarations in the block (not shown) that immediately contains this procedure:

- 'REP' with attributes 'entry(char(1000) varying, fixed) returns(char(1000) var) internal constant'
- 'REP2' with attributes 'entry(char(1000) varying) returns(char(1000) varying) internal constant'

The attributes for 'REP' were obtained by making two changes in the 'procedure' statement and adding the attributes 'internal constant'. The changes are the replacement of 'procedure' by 'entry' and the replacement of each parameter by the declaration of the parameter. The attributes for 'REP2' were obtained by a similar modification of the 'entry' statement. The declarations are established in the containing block because a procedure block does not contain the label prefixes of 'procedure' and 'entry' statements in the procedure block.

The example just given shows that the declaration of an entry constant name can be long. Its purpose is to supply information needed to convert values supplied as arguments and to accept the value produced as a result. More is said of this later, in Section XII, "Procedure Invocation."

Suppose the procedure just given is not contained in some larger block (as was previously assumed) but is itself an external procedure. This change has two effects. First, the declarations established for both 'REP' and 'REP2' are changed by replacing 'internal' by 'external'. Second, the declarations of 'REP' and 'REP2' are established in the imaginary 'begin' block that encloses the program.

FORMAT CONSTANT NAMES

When a label prefix occurs before a 'format' statement, the identifier in the prefix is a format constant name. An effect of the label prefix is to establish in the immediately containing block a declaration of an identifier with the attributes 'format internal constant'. The role of 'format' statements in PL/I is rather limited and is described under "Stream Input/Output."

Contextual and Implicit Declarations

A declaration that is established by a 'declare' statement or a label prefix is called an explicit declaration. PL/I allows declarations to be established in other ways, and these declarations are contextual or implicit. A contextual declaration is one that is indicated by the way in which an identifier is used; an implicit declaration is a last resort used when no other basis for establishing a declaration can be found.

EXAMPLE OF CONTEXTUAL AND IMPLICIT DECLARATIONS

An example of a program that uses contextual and implicit declarations is:

```
A:  proc;
      do i = 0 to 90;
          put file(sysprint) skip list(sind(i));
      end;
end;
```

This program fails to give explicit declarations for 'sysprint', 'sind', and 'i'. The PL/I compiler assumes that declarations for these names should be established in the outermost block of the external procedure (the only block in this case) and supplies attributes as follows:

- 'sysprint' occurs in a 'file' option; since only a 'file' value can occur in this context, the compiler supplies the attributes 'file constant' as a contextual declaration.
- 'sind' occurs as a function or array name; since 'sind' is the name of a built-in function, the compiler supplies the attribute 'builtin' as a contextual declaration.
- 'i' occurs in a context which does not unambiguously indicate its declaration; therefore the compiler supplies no attributes, and the declaration is implicit. According to the default rules, a name with no attributes is assumed to be 'real fixed binary (17,0)'.

It follows that the example program is equivalent to the following program, in which all identifiers are explicitly declared:

```
A:  proc;
      dcl sysprint file;
      dcl sind builtin;
      dcl i fixed bin;
      do i = 0 to 90;
          put file(sysprint) skip list(sind(i));
      end;
end;
```

GUIDELINES FOR CONTEXTUAL AND IMPLICIT DECLARATIONS

The example just given shows that contextual and implicit declarations make a short PL/I program much shorter. However, in a larger, more realistic program, the use of contextual or implicit declarations can mask errors in the program. Contextual and implicit declarations are implemented in Multics PL/I and mentioned here because they are part of Standard PL/I. However, the Multics PL/I compiler prints a warning message for each such declaration, and it is recommended that every name in a Multics PL/I program be explicitly declared by 'declare' statement or a label prefix.

Special Facilities for Declaration

PL/I has two facilities that are designed to assist programmers in the declaration of names: the 'like' attribute and the 'default' statement. The 'like' attribute is recommended for the rather special situations to which it applies, and it is fully described here. In contrast, the 'default' statement is not recommended, and only a few examples are given here rather than a full definition.

'like' ATTRIBUTE

The 'like' attribute asserts that the members of a given structure have the same declarations as the members of some other structure. For example, consider the 'like' attribute in the following program:

```
P:  proc;
    dcl 01 alpha(1000) external static,
        02 x float bin,
        02 y char(16);
    dcl 01 beta like alpha;
    ...
end;
```

The second 'declare' statement is equivalent to:

```
dcl 01 beta,
    02 x float bin,
    02 y char(16);
```

Form of the 'like' Attribute

The 'like' attribute has the following form:

like lr

where lr is the like reference. The like reference must be a name or a sequence of names separated by periods. A use of the 'like' attribute must satisfy the following restrictions:

- The 'like' attribute can be used only in a 'declare' statement.
- The 'like' attribute can apply only to a structure name; that is, it must appear in a declaration clause that begins with a level number.
- A structure declaration with a 'like' attribute must not be followed by a member declaration. That is, if the declaration clause that contains the 'like' attribute has level number n, then the immediately following declaration clause must not have a level number that is greater than n.
- It must be possible to resolve the like reference in a 'like' attribute according to the rules for the resolution of name references given later in this section.
- The structure designated by the like reference must not contain a 'like' attribute. That is, a 'like' attribute cannot be defined in terms of some other 'like' attribute.

Interpretation of the 'like' Attribute

A 'like' attribute is interpreted by the compiler. First, the like reference is resolved; the result is the declaration of a structure. The designated declaration consists of a declaration clause for the structure name itself followed by a sequence of declaration clauses for the members of the structure, the members of the members of the structure, and so on. The compiler copies the sequence of member declaration clauses into a position immediately after the declaration clause that contains the given 'like' attribute; then it deletes the 'like' attribute. The final result is a complete declaration of a structure.

The designated declaration clauses are copied literally, before any attributes are filled in by the various default mechanisms of PL/I. The level numbers are adjusted, if necessary, to assure that the declaration clause for a member has a higher number than that for the containing structure. Within a given block, the result of interpreting one 'like' attribute is not used in interpreting another 'like' attribute; such possibilities are ignored when a like reference is resolved.

Examples of the 'like' Attribute

The following program contains three examples of the use of the 'like' attribute:

```
P:  proc;
    dcl 01 customer(1000) external controlled,
        02 ident,
        03 name(3) char(30),
        03 number pic"999b99b9999",
        02 balance dec(8,2);
    dcl 01 current based like customer;
    dcl 01 ident_list(20) like customer.ident;
    dcl 01 ident_pair external static,
        02 old like customer.ident,
        02 new like customer.ident;
    ...
end;
```

The last three 'declare' statements are interpreted as follows:

```
dcl 01 current based,
    02 ident,
    03 name(3) char(30),
    03 number pic"999b99b9999",
    02 balance dec(8,2);
dcl 01 ident_list(20),
    02 name(3) char(30),
    02 number pic"999b99b9999";
dcl 01 ident_pair external static,
    02 old,
    03 name(3) char(30),
    03 number pic"999b99b9999",
    02 new,
    03 name(3) char(30),
    03 number pic"999b99b9999";
```

Guidelines for the 'like' Attribute

A 'like' attribute should not be used merely to save writing; it should be used only when there is a close relationship between structure variables.

THE 'default' STATEMENT

A 'default' statement is a rule for adding attributes to the declaration of a name, a constant literal, a parameter, or the returned result of a procedure. The statement applies to every such declaration within its scope.

A 'default' statement is composed of a default test, which examines the attributes already present in a given declaration, and a sequence of default attributes that are added to the given declaration when the default test is satisfied. A 'default' statement is fully interpreted by the compiler, and it has no direct action when a program is executed.

The definition of the 'default' statement is not given in this manual; it appears in the PL/I Language Manual. A few examples are given here, however, in order to provide a brief introduction to the statement.

Examples of the 'default' Statement

As an example of the use of a 'default' statement, suppose it is necessary to use double precision for binary floating-point values in a certain block. The following statement can be used to achieve the desired effect by introducing a new default for the number-of-digits:

```
default (float & ^dec & ^prec) prec(63);
```

This statement means "wherever a 'float' attribute is present and a 'decimal' attribute is not present and a 'precision' attribute is not present, insert the 'precision(63)' attribute".

The example just given requires some discussion. Why not use 'bin' instead of '^dec'? And why use '^prec' at all? The answer is that a default test must be written very carefully to cover all possible cases. Consider the following statement:

```
dcl x float bin;
```

Even though the system defaults will eventually add 'binary' to this declaration, they are applied after, not before, the 'default' statement is applied. Therefore, the use of 'bin' in the default test instead of '^dec' would miss this declaration of 'x'. Next, consider:

```
dcl y float prec(30);
```

A 'default' statement adds an attribute rather than replacing an attribute. If the '^prec' is omitted from the default test, then the 'default' statement applies to this declaration of 'y' and the result is:

```
dcl float prec(30) prec(63);
```

This is an invalid 'declare' statement.

A 'default' statement can be used to exclude certain declarations; for example:

```
default (complex) error;
```

This statement means "wherever a 'complex' attribute is present, the program is in error". When the default test is satisfied for this 'default' statement, the compiler prints a diagnostic message. This example suggests that the 'default' statement could be used to enforce the use of a subset of PL/I. However, the 'default' statement is too limited to cover many such cases. For example, there is no way to require that a 'fixed binary' value have a zero scale factor.

As a third example of a 'default' statement, consider the following:

```
default(^ (range(i:n)|range(I:N)
          & ^ (constant|builtin|condition|generic)
          & ^ (character|bit|pointer|offset|area|label|entry|file)
          & ^ (fixed|decimal|precision))

float binary prec(27);
```

This statement is complicated because it must cover all possible cases; however, it means that an arithmetic variable that does not begin with IJKLMN (upper or lower case) is assumed to be 'float binary precision(27)' when attributes to the contrary are not given. When used in conjunction with the system defaults, it approximates the handling of variable names in FORTRAN.

Guidelines for the 'default' Statement

The examples of the 'default' statement just given show that it is difficult to use. The problems with the 'default' statement can be summarized as follows:

- It is too deep. The 'default' statement is applied after certain special defaults are applied and before the standard system defaults are applied; furthermore, the order in which several 'default' statements are written can affect their results.
- It is too limited. The test in a 'default' statement is not powerful. Many useful defaults cannot be programmed and others can be programmed only in an indirect and complicated way.
- It is unnecessary. PL/I already has an elaborate set of standard system defaults. A departure from those defaults introduces unwelcome complications.

For these reasons, the use of the 'default' statement is not recommended.

RESOLUTION OF NAME REFERENCES

This discussion of the resolution of name references begins with an example; then the specific definitions and rules for resolution are given.

Example of the Resolution of Name References

For examples of the resolution of names, consider once again the following program, which was given at the beginning of this section:

```
P:  proc;
    decl x float bin;
    decl (sysin,sysprint) file;
L2: get list (x);
    call Q;
    if x = 0 then goto L2; else return;
Q:  proc;
    decl L2 float bin;
    L2 = x**2 - 3*x**2;
    put skip list(x,L2);
    end;
end;
```

Consider the five instances of the name 'L2' in this program. One instance is in a label prefix and another is in a 'declare' statement; these establish declarations for 'L2'. Three instances of 'L2' remain, and these instances are name references that are resolved as follows:

- The name reference in the 'goto' statement is resolved to the declaration of 'L2' that is established in the outer block; therefore, this name reference is associated with the outer block and has attributes 'label internal constant'.
- The name references in the assignment statement and the 'put' statement are resolved to the declaration of 'L2' that is established in the inner procedure; therefore, each of these name references is associated with the inner block and has the attribute 'float'.

The rules under which this resolution was performed are given later in this section. First, however, some definitions must be given.

Name-Sequence Set for a Declaration

Each declaration has an associated set of name sequences. If the declaration describes a structure variable, then the set contains the level-one name of the structure and also contains each sequence of names that is formed by starting with the level-one name and proceeding through contained level names. If the declaration does not describe a structure variable, then the set contains just one name-sequence and that name sequence is the declared name.

Four examples of declarations and their associated sets of name sequences are given in the following table:

<u>Declaration</u>	<u>Set of Name Sequences</u>
dcl 01 Q(0:5) based, 02 R1 float bin; 02 R2 fixed dec(8,2);	Q Q.R1 Q.R2
dcl 01 S static external, 02 Wef(2,m-3), 03 P float bin, 03 Q, 04 rho(n) fixed dec(4), 04 phi fixed bin, 03 R float bin, 02 G(10,10,cnt) float bin;	S S.Wef S.Wef.P S.Wef.Q S.Wef.Q.rho S.Wef.Q.phi S.Wef.R S.G
dcl alpha(n+2) float bin static;	alpha
dcl sqrt builtin;	sqrt

Name-Sequence for a Name Reference

Each name reference has an associated name sequence. If the name reference is a structure-qualified variable reference, then the associated name sequence is the sequence of level names in the reference. In all other cases, the name sequence is just the name itself.

Four examples of name references and their associated name sequences are given in the following table:

<u>Name Reference</u>	<u>Name Sequence</u>
Q(3*n1+n2).R1	Q.R1
S.Wef(8,m-3).Q	S.Wef.Q
alpha(2*i-beta)	alpha
sqrt	sqrt

The full definition of structure-qualified variable references is given later, in Section VIII, "Expressions."

Applicability of Declarations

A declaration can be applicable to a given name reference in two ways, as follows:

- The declaration is applicable if it has a name sequence that is identical to the name sequence for the given name reference. In this case, the name reference is fully-qualified with respect to the declaration.

- The declaration is applicable if it has a name sequence that, after the omission of one or more names, becomes identical to the name sequence for the given name reference. If only this case applies, then the name reference is partially-qualified with respect to the declaration.

As a basis for some examples of applicability, consider the following declaration of the structure 'omega':

```
dcl 01 omega controlled,
    02 m,
    03 S1 char(30),
    03 S2 fixed bin,
    02 gamma float bin;
```

A complete list of the references to which this declaration is applicable follows:

<u>Fully-Qualified</u>	<u>Partially-Qualified</u>
omega	m
omega.m	omega.S1
omega.m.S1	m.S1
omega.m.S2	S1
omega.gamma	omega.S2
	m.S2
	S2
	gamma

Resolution Rules

To resolve a given name reference, begin the search at the block that contains the given reference and continue the search outward for each containing block to find a block that has an established declaration that is applicable to the given use of the name reference. There are three possibilities, as follows:

- No such block is found. In this case, the name reference is undeclared. The use of an undeclared name reference is not necessarily an error in Standard PL/I; a declaration is supplied according to the rules for contextual and implicit declaration, mentioned earlier in this section. However, the use of an undeclared name reference is not recommended in Multics PL/I and the compiler marks such a use with a warning.
- Exactly one such block is found. In this case, that is the desired block and resolution proceeds as in the next paragraph.
- More than one such block is found. In this case, the desired block is the smallest of the blocks, and resolution proceeds for that block according to the next paragraph.

When the desired block has been found, there are three possible cases to be considered, as follows:

- If the block contains exactly one applicable declaration, then that declaration is the declaration of the given name reference and resolution is complete.

- If the block contains more than one applicable declaration, but only one for which the given name reference is fully-qualified, then that one is the declaration of the name reference and the resolution is complete.
- If the block contains more than one applicable declaration but the preceding case does not apply, then the declaration is ambiguous and the name reference is invalid.

As a basis for examples of name resolution, consider the following program:

```

P:  proc;
    dcl X float bin;
    dcl Y float bin;
    ... (Computation #1)
Q:  proc;
    dcl 01 S,
        02 Y float bin,
        02 Z fixed dec(8,2);
    dcl 01 R,
        02 alpha char(16),
        02 Z(100) float bin;
    dcl alpha fixed bin;
    ... (Computation #2)
    end;
end;

```

Now consider some of the references that could appear in Computation #1 or Computation #2:

- X In either Computation, this reference is a fully-qualified reference to the 'X' declared by the first 'declare' statement in the outer block.
- Y In Computation #1, this reference is a fully-qualified reference to the 'Y' declared in the second 'declare' statement in the outer block. In Computation #2, this reference is a partially-qualified reference to the first member of 'S' declared in the first 'declare' statement in the inner block.
- alpha In Computation #1, this reference is undeclared and, in Multics, is contrary to recommended usage. In Computation #2, this reference is a partially-qualified reference to the first member of 'R' and a fully-qualified reference to the 'alpha' that is declared in the last 'declare' statement in the inner block. The reference is resolved to the second possibility, the 'alpha' declared in the last 'declare' statement.
- Z(i) In Computation #1, this reference is undeclared. In Computation #2, this reference is a partially-qualified reference to the second member of both 'S' and 'R'. Both 'S' and 'R' are declared in the inner block; therefore, the reference cannot be resolved and is invalid.

Observe that the subscript in the reference 'Z(i)' does not enter into the resolution of the reference, even though 'R.Z' can have a subscript and 'S.Z' cannot.

ATTRIBUTES

PL/I has more than 50 attributes, and many combinations of these can be used in the declaration of a name. The following paragraphs present two views of attributes. First, five kinds of names are defined and the sets of attributes allowed for the declaration of each kind of name are given. Then a complete classification of the attributes is presented.

Complete Attribute Sets

There are five main kinds of names, as follows:

- variable names
- constant names
- built-in function names
- condition names
- generic names

The different kinds of names vary widely in their importance. The variable names have a variety and flexibility that overshadows all other names. Constant names and built-in function names appear in most programs. Condition names also occur in most programs, but they are used in a restricted context. Generic names are extremely specialized and are rarely used.

The following paragraphs give the complete attribute sets for each kind of name. A set of attributes is a valid complete declaration for a name if and only if it is included in one of these sets. As an example, consider one of the complete attribute sets for a variable name:

real float binary precision(27) aligned automatic internal variable

Because of the default rules of the language, this attribute set can be shortened to:

float

However, the following paragraphs do not mention such shortened forms. They give only complete attribute sets, before any abbreviations or defaults have been applied. In practice, once an appropriate attribute set has been determined, it is a relatively routine job to apply abbreviations and defaults to shorten it.

The complete attribute sets are given by means of diagrams. The diagrams use two special notations, as follows:

- Curly braces indicate a choice; they enclose two or more lines, any one of which is chosen in making up a specific attribute set.
- Square brackets indicate an option; they enclose an item that can be either included or omitted in making up a specific complete attribute set.

Some of the identifiers in the diagrams are underlined and some are not. The underlined identifiers are terms that are defined in the text that follows the diagram. The nonunderlined identifiers are attribute keywords.

VARIABLE NAMES

A variable name designates storage for a value obtained from input or calculation. The complete attribute sets for a variable name are:

$$\underline{dt} \left[\text{dimension}(\underline{bp}, \dots) \right] \left\{ \begin{array}{l} \text{aligned} \\ \text{unaligned} \end{array} \right\} \underline{sc} \left[\text{initial}(\underline{x}, \dots) \right] \text{variable}$$

where dt is the data type, bp,... is a sequence of array-bound pairs separated by commas, sc is the scope and class, and x,... is a sequence of initial value expressions separated by commas.

The data type is one of the following sets of attributes:

$$\left\{ \begin{array}{l} \left\{ \begin{array}{l} \text{real} \\ \text{complex} \end{array} \right\} \quad \left\{ \begin{array}{l} \text{fixed} \\ \text{float} \end{array} \right\} \quad \left\{ \begin{array}{l} \text{binary} \\ \text{decimal} \end{array} \right\} \quad \left\{ \begin{array}{l} \text{precision}(\underline{p}, \underline{q}) \\ \text{precision}(\underline{p}) \end{array} \right\} \\ \\ \text{picture} \underline{ps} \quad \left\{ \begin{array}{l} \text{real} \\ \text{complex} \end{array} \right\} \\ \\ \left\{ \begin{array}{l} \text{character}(\underline{ee}) \\ \text{bit}(\underline{ee}) \end{array} \right\} \quad \left\{ \begin{array}{l} \text{nonvarying} \\ \text{varying} \end{array} \right\} \\ \\ \text{label} \quad [\text{local}] \\ \text{entry}(\underline{d}, \dots) \quad [\text{options}(\text{variable})] - [\text{returns}(\underline{d})] \\ \text{format} \quad [\text{local}] \\ \text{pointer} \\ \text{offset} \quad [(\underline{a})] \\ \text{file} \\ \text{area} \quad [(\underline{ee})] \\ \text{structure} \quad [\text{like } \underline{r}] \end{array} \right\}$$

where p is an unsigned decimal integer constant, q is an optionally signed decimal integer constant, ps is a picture, ee is an expression whose value can be converted to an integer value, d is a descriptor, a is a reference that yields an 'area' value, and r is a like-reference.

The scope and class is one of the following sets of attributes:

$$\left(\begin{array}{l} \left\{ \begin{array}{l} \text{static} \\ \text{controlled} \end{array} \right\} \quad \left\{ \begin{array}{l} \text{internal} \\ \text{external} \end{array} \right\} \\ \\ \left\{ \begin{array}{l} \text{automatic} \\ \text{based(} \underline{lg} \text{)} \\ \text{parameter} \\ \text{defined(} \underline{br} \text{)} \quad [\text{position(} \underline{i} \text{)}] \\ \text{member} \end{array} \right\} \quad \left. \vphantom{\left\{ \begin{array}{l} \text{automatic} \\ \text{based(} \underline{lg} \text{)} \\ \text{parameter} \\ \text{defined(} \underline{br} \text{)} \\ \text{member} \end{array} \right\}} \right\} \text{internal} \end{array} \right)$$

where lg is a locator qualifier, br is a based reference, and i is an expression whose value can be converted to an integer value.

Variable names are described in Section VIII, "Expressions."

CONSTANT NAMES

A constant name designates a statement address or a file-state-block address. The address is set by the compiler and does not change during program execution. The complete attribute sets for a constant name are:

$$\left(\begin{array}{l} \text{label internal } [\text{dimension(} \underline{bp} \text{)}] \\ \\ \text{entry(} \underline{d}, \dots \text{)} \quad \left\{ \begin{array}{l} \text{internal} \\ \text{external} \end{array} \right\} \quad [\text{options(variable)}] \quad [\text{returns(} \underline{d} \text{)}] \\ \\ \text{format internal} \\ \\ \text{file} \quad \left\{ \begin{array}{l} \text{internal} \\ \text{external} \end{array} \right\} \quad \underline{fd} \end{array} \right) \text{constant}$$

where bp is a pair of array bounds, d is a descriptor for a parameter or a result, and fd is the file description, defined as follows:

$$\left(\begin{array}{l} \text{stream} \quad \left\{ \begin{array}{l} \text{input} \\ \text{output} \quad \text{print} \quad [\text{environment(interactive)}] \end{array} \right\} \\ \\ \text{record} \quad \left\{ \begin{array}{l} \text{input} \\ \text{output} \\ \text{update} \end{array} \right\} \quad \left\{ \begin{array}{l} \text{sequential} \\ \text{sequential} \\ \text{direct keyed} \end{array} \right\} \quad \text{environment(stringvalue)} \end{array} \right)$$

Constant names are described in Section VIII, "Expressions."

BUILT-IN FUNCTION NAMES

A built-in function name designates an operation that is applied to arguments to produce a result. Each built-in function name designates an operation whose action is a fixed part of the definition of PL/I. The complete attribute set for a built-in function name is:

internal builtin

Built-in function names are described in a general way in Section VIII, "Expressions," and the individual built-in functions are defined in Section IX, "Operations."

CONDITION NAMES

A condition name designates an exceptional situation that can arise during program execution; it is used in programming a response to the given situation. The complete attribute set for a condition name is:

external condition

Condition names are described in Section XIII, "Condition Handling."

GENERIC NAMES

A generic name designates a set of rules for selecting a programmed entry name from a set of programmed entry names. The compiler follows the rules and replaces the generic name with one of the programmed entry names. The complete attribute set for a generic name is:

internal generic(alt,...)

where 'alt,...' is a sequence of alternatives separated by commas. Generic names are described in Section XII, "Procedure Invocation."

Classification of Attributes

It is useful to arrange the attributes in a single, hierarchical classification and thus establish a complete terminology for the discussion of attributes. The classification is as follows:

storage description

storage type

data type

computational

arithmetic

mode: real complex
scale: fixed float
base: binary decimal
precision: precision(p,q)

string

string type: character(ee) bit(ee) picture"ps"
variability: varying nonvarying

non-computational

address

statement: label entry format

data

locator: pointer offset
file: file

area: area(ee)

aggregate type

array: dimension(bp,...)
structure: structure member
alignment: aligned unaligned

management class

storage class

allocation: automatic static controlled based(lq)
sharing: based(lq) defined(r) position(i) parameter
scope: internal external
category: variable constant
initial: initial(x,...)

usage description

label and format: local

entry: entry(d,...) returns(d,...) options(variable)

offset: offset(a)

file constant

operation: input output update

organization

stream: stream print environment(interactive)
record: record sequential direct keyed
environment (string value)

non-valued names

compile-time: like r generic(alt,...)

intrinsic names: builtin condition

Definitions for the underlined identifiers, p, q, ee, and so on, are not given here because they were given earlier, in the definitions of the complete attribute sets. Further information about a given attribute or class of attributes can be located through the index of this manual.

SECTION VII

STORAGE MANAGEMENT

Part of the cost of program execution is expended on the computing resource called storage. When the storage requirement can be reduced, the cost of the program execution decreases. A programmer cannot do much to reduce the storage occupied by a given program, but he can exercise some control over the amount of storage required for the data on which the program operates. The control of data storage is called storage management. Observe that storage management concerns storage itself, and not the values contained in that storage.

Each variable name in a program must have storage allocated for its value at some time before it is assigned a value and may have that storage freed at some time after the last use of its value. Allocation and freeing are the fundamental operations of storage management. There are several mechanisms that cause these operations to occur, and the programmer chooses one mechanism for each variable name by giving a storage class attribute when he declares the variable name.

The requirement of a program for storage can be reduced by storage management; the effect is achieved by using a given portion of storage for more than one purpose and thus "recycling" the storage resource. However, storage management has its own cost because it is, in itself, a form of data processing. In some cases, the saving of storage does not justify the increase in processing cost and program complexity. In Multics, small scale storage management is not recommended. Programmed storage management is usually reserved for cases in which a large saving in storage can be accomplished in a simple way.

Some of the principal innovations of both Multics and PL/I are applied to the problem of storage management. The Multics paging system is a hardware-based mechanism for storage management that is automatic and quite invisible to the programmer. The block structure of PL/I allows the programmer to organize programs in a way that implies the desired storage management without requiring explicitly programmed allocation and freeing of storage. In addition, PL/I has built-in routines that facilitate storage management under program control. Taken together, these features of Multics and PL/I allow the programmer to use very powerful storage management techniques with a small amount of programming effort.

This section has three main parts. The first part gives examples and the fundamentals of storage allocation. The second part defines the management class, which consists of the usage, scope, storage class, and initial attributes. The final part discusses the capacity of storage and the exceptional conditions that apply to storage allocation.

PRELIMINARY EXAMPLES OF STORAGE MANAGEMENT

A few concrete examples are given here to prepare for the detailed discussion of storage management. Consider, first, a program that has a minimum of storage management:

```
P1:  proc;
      dcl (sysin,sysprint) file;
      dcl (a,b,c) float bin static;
      get list(a);
      b = a+4;
      c = b**2;
      put list(b,c);
      end;
```

Because the variable names 'a', 'b', and 'c' are declared 'static', their storage is allocated throughout the process from the first reference to the program 'P1' on. Since there are times when a variable is not in use (that is, does not contain a useful value), storage is wasted.

One way of introducing storage management is to use 'controlled' variables. Then statements can be inserted that allocate each variable just before its first use and free it just after its last use, as in the following revision of the given example:

```
P2:  proc;
      dcl (sysin,sysprint) file;
      dcl (a,b,c) float bin controlled;
      allocate a;
      get list(a);
      allocate b;
      b = a+4;
      free a;
      allocate c;
      c = b**2;
      put list(b,c);
      free b,c;
      end;
```

The digit at the end of each line is not part of the program; it is added to show how many 'float bin' variables are in storage after each statement. Instead of using three variables all the time, the program never uses more than two. On the other hand, the program has more statements, and these statements add to the cost of executing the program. The version just given could not be justified to save one or two 'float bin' variables; but it might be appropriate if large arrays, for example, were involved.

An entirely different approach can be taken to reducing the storage required for the given program. The programmer can recognize that the last use of the variable 'a' occurs before the first use of the variable 'c'; this, of course, is a special property of this particular calculation. The programmer could then replace each occurrence of 'c' by 'a' throughout his program and omit the declaration of 'c'; but this would confuse the logic of the program (since 'a' and 'c' represent different mathematical quantities). A different approach is to use a 'defined' variable, as follows:

```
P3: proc;
    dcl (sysin,sysprint) file;
    dcl (a,b) float static;
    dcl c float bin defined(a);
    get list(a);
    b = a+4;
    c = b**2;
    put list(b,c);
end;
```

The 'defined' attribute causes 'c' to occupy the same storage as 'a', so the program needs storage only for two 'float bin' variables. This storage management technique has virtually no cost in terms of execution; however, it requires an analysis of the program, and an error in that analysis can easily lead to costs far in excess of the saving in storage.

Both of these examples of programmed storage management have weaknesses: the first incurs a considerable processing cost for allocation and freeing and the second is trick programming and is not recommended. Such techniques are more appropriate when PL/I is used to program a computer with a small storage capacity. There are occasions when programmed storage management is necessary in Multics PL/I programs; but most Multics PL/I programs rely entirely on the allocation and freeing performed by PL/I in response to the structure of the program.

FUNDAMENTALS OF STORAGE MANAGEMENT

A Multics PL/I programmer does not deal directly with hardware storage; instead, he works through several layers of special hardware and software whose purpose is to simplify the use of storage. These levels begin at the hardware, continue with the virtual memory, and conclude with the PL/I regions.

- On the hardware level, storage is a small working memory and a large secondary memory. Such a configuration would be difficult to program, since it requires an almost constant transmission of data and instructions between the working storage and the secondary storage. The Multics paging system controls this transmission and makes all storage appear to be a homogeneous memory, called virtual memory.
- On the virtual memory level, storage is a set of segments. Each segment is an addressable memory of 36-bit words and can be used like the working storage of an ordinary processor. However, a segment is very large (over 200,000 words) and, furthermore, the number of segments is very large. Thus virtual memory is an idealization of the actual hardware storage.

- On the PL/I language level, storage is thought of as a set of regions that are mapped in fairly simple ways into the segments of virtual memory. Each region has a clearly defined relation to the PL/I program with which it is associated, and regions are created and destroyed as an integral part of program execution. Thus PL/I storage is a specialization of virtual memory.

Usually a PL/I programmer thinks entirely in terms of storage regions; that is, he considers these regions to be the "real" data storage of a PL/I program. A concern for the way in which regions are laid out in segments and distributed between working and secondary storage is necessary only for considerations of cost and debugging of invalid programs.

Storage Regions

Each storage region is a separate entity with a fixed but very large capacity. A region is either internal or external. Each internal region is associated with a particular block of a program. There are two kinds of internal regions:

- A permanent internal region is used for variables with 'internal static' or 'internal controlled' attributes. There is one such region for each block in a program. The region is created at some time before program execution begins and is not destroyed until after the process ends.
- An activation internal region is used for variables with 'internal automatic' attributes. There is one such region, called the stack frame, for each activation of each block in a program. The region is created when the block is activated and is destroyed when the block is deactivated.

An external region is associated with an entire process rather than a particular block. There is one external region:

- The ordinary external region is used for variables and constants that have the attribute 'external' and that do not have special names. (A special name is an identifier that contains a '\$'.) There is one such region for an entire process. The region is created at some time before program execution begins and is not destroyed until after the process ends.

To summarize, the regions are classified according to the following hierarchy:

```

region
  internal
    permanent: one per block
    activation: one per activation of each block
  external
    ordinary: one per process

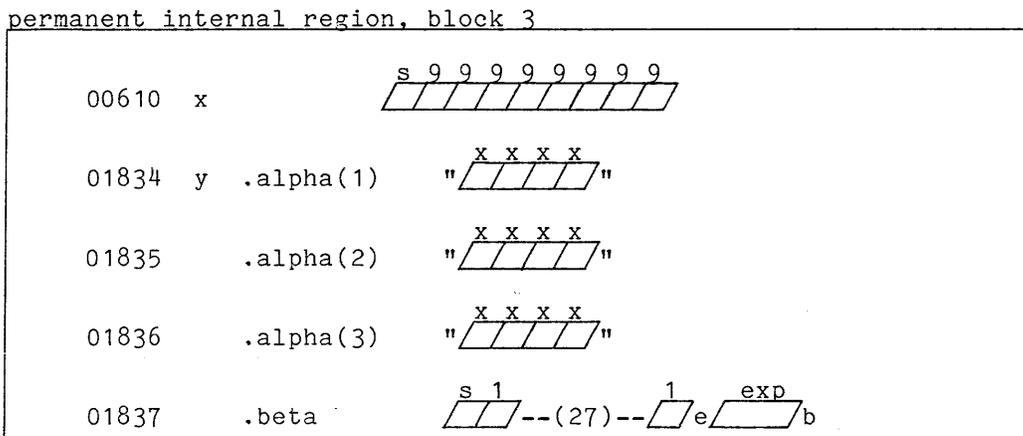
```

Region Diagrams

For purposes of discussion, it is useful to represent a region by a diagram. Suppose the following statements are all of the declarations in the third block of some given program:

```
begin;
  dcl x fixed dec(8) static internal;
  dcl 01 y static internal,
      02 alpha(3) char(4),
      02 beta float;
  ...
end;
```

The program in question has a set of regions associated with it. The diagram of one of these regions is:



This region is the permanent internal region for the third block of the program. It contains five storage units, one for the scalar variable 'x' and four more for a structure variable 'y'. The region continues downward, as suggested by the absence of a bottom edge for the enclosing boundary, and has room for many more storage units.

Each line of the region describes a storage unit. A line is made up of four components: a representation of a pointer value, a name, an access path composed of member names and subscripts, and a data frame.

Storage Management Operations

At any time during the execution of a program, each bit of storage in a region is either allocated or free. Storage is allocated if a designation that can be used in program execution is associated with it; otherwise, it is free. Free storage is organized into a pool from which requests for storage are satisfied.

The purpose of a storage management operation is to control the association between a designator (most commonly, a variable name) and its storage. The allocation operation for a variable name can be invoked either by the PL/I processor or by a program statement; but in either case, the details of the operation are determined by the attributes given in the declaration of the name. The allocation operation proceeds as follows:

1. Each extent expression (array bound, maximum string length, or area size) in the storage type attributes is evaluated and converted to an integer value. This evaluation is performed each time the allocation operation is performed.
2. The amount of storage required is determined from the given storage type attributes and their evaluated extents.
3. The region in which allocation must occur is determined from the given storage class attributes. Storage is withdrawn from the free storage pool of that region and is associated with the given variable name.
4. If the variable is an 'area' variable, it is initialized to empty. If the variable has an 'initial' attribute, it is initialized; any expressions in the 'initial' attribute are evaluated each time the allocation operation is performed.

The free operation takes storage back from a given storage designator and returns it to the pool of free storage.

Storage Management Example

Consider the following program, which is made up of two external procedures, 'P1' and 'P2':

```
P1:  proc;
      dcl P2 entry external constant;
      dcl A dec(5,2) controlled external;
      dcl B dec(5,2) static internal;
      ... (Computation #1)
      call P2;
      ... (Computation #2)
      free A;
      end;
```

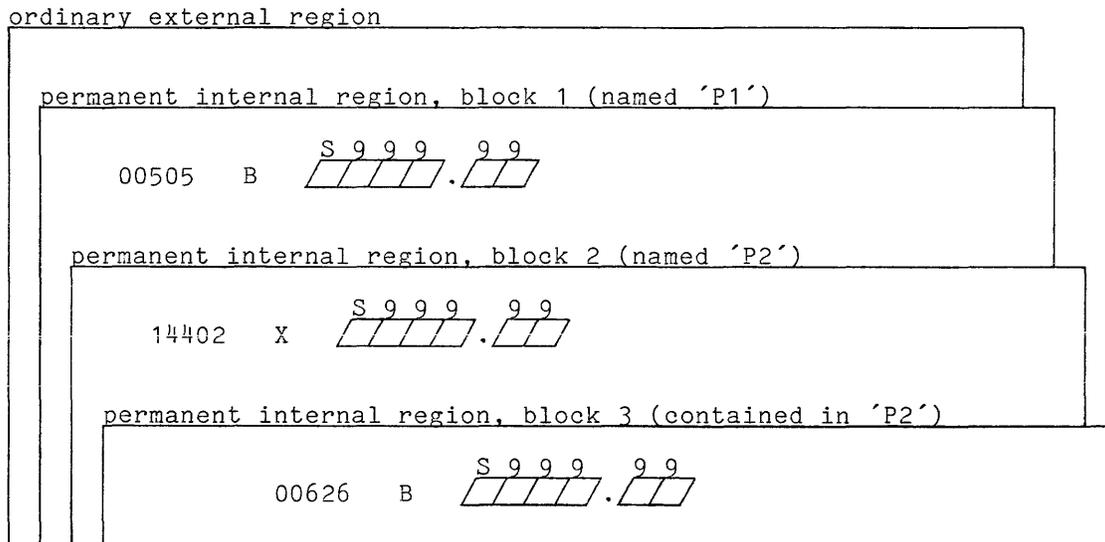
```

P2: proc;
    dcl X fixed dec(5,2) static internal;
    ... (Computation #3)
    begin;
    dcl A fixed dec(5,2) controlled external;
    dcl B fixed dec(5,2) static internal;
    dcl Y fixed dec(5,2) automatic internal;
    ... (Computation #4)
    allocate A;
    ... (Computation #5)
    end;
end;

```

The "Computations" in this program represent statements that do not introduce new declarations or affect the block structure of the program. Some of the attributes shown would normally be omitted by means of default conventions; to omit them here would not serve the purposes of an introductory example.

The following discussion traces the execution of this program and describes each allocation operation as it occurs. Before execution begins, a single ordinary external region is created; for this program, it is initially empty. Also before execution begins, a permanent internal region is created for each block and each 'static internal' variable is allocated. Immediately before execution of the program, the diagram for storage is:

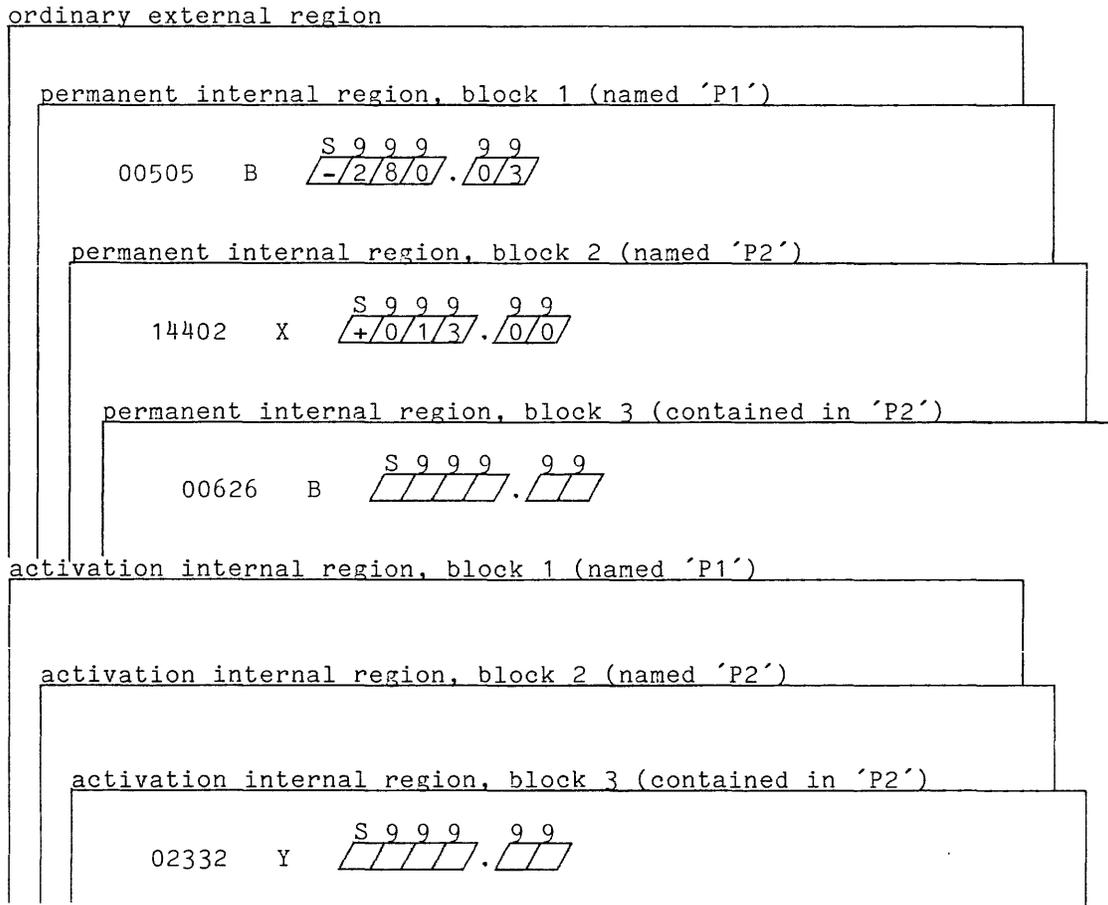


The first step in the execution of the program is the activation of the block that is the external procedure 'P1'. As part of this activation, an internal activation region is created; but because there is no automatic variable in the block, no storage unit is allocated in the new region.

Execution continues with Computation #1. It is assumed that the only allocation of the controlled variable 'A' is the one shown in 'P2'; therefore, the only storage unit available for use so far is the static variable 'B'. After the computation, the procedure 'P2' is called.

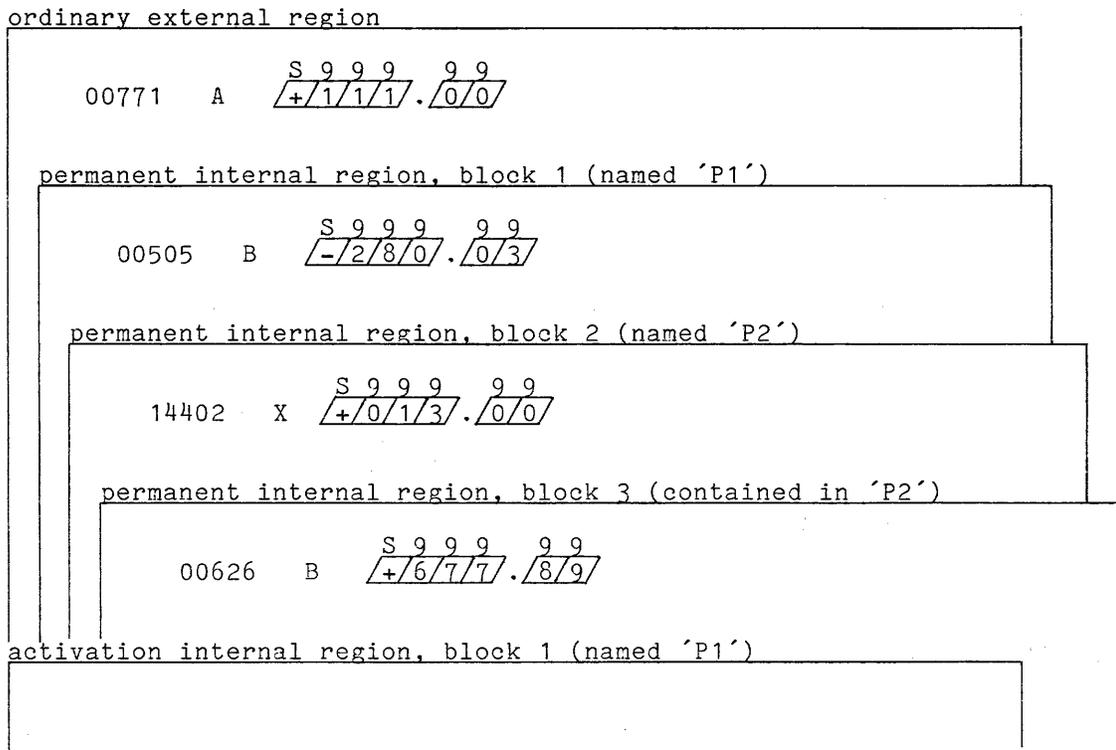
The first step in the execution of 'P2' is the activation of the block; an internal activation region is created, but once again there is no automatic variable in the block, so no storage unit is allocated in the new region. Execution continues with Computation #3, which can use only the static variable 'X'. After the computation, control flows into the 'begin' block.

The execution of the 'begin' block begins with its activation. Again, an internal activation region is created; but now there is an automatic variable, 'Y', in the block and it is allocated as part of the activation. As Computation #4 is about to begin, the diagram for storage is:



Computation #4 can use the static variable 'B' and the automatic variable 'Y'. Observe, however, that the 'B' that is accessible is the one in the third permanent internal region, not the one in the first. Thus in this example, 'P2' cannot use any value that was computed in 'P1'.

After Computation #4 is executed, a statement is executed that allocates a storage unit for the controlled variable 'A' in the ordinary external region. During Computation #5, a value is stored in 'A', and this, it is assumed, is a useful result that 'P2' prepared for use by 'P1'. As execution of the 'begin' block and 'P2' is completed, their regions are discarded; then control returns to 'P1'. As Computation #2 is about to begin, the diagram for storage is:



Computation #2 can use the static variable 'B' allocated for block 1; in addition, now that a storage unit has been allocated for 'A', this variable can be used in the computation.

After Computation #2, a 'free' statement returns the storage for 'A' to the free storage pool. In the storage diagram, the ordinary external region once again is empty. The last step in execution of the program is the deactivation of the block that is external procedure 'P1'; as part of this step, the internal activation region is discarded. At this time, the storage diagram looks just as it did before program execution (see the first diagram of this series) except that the stored values are filled in. When the process of which the program is a part ends, the remaining regions are discarded, and no storage associated with the program remains.

MANAGEMENT CLASS

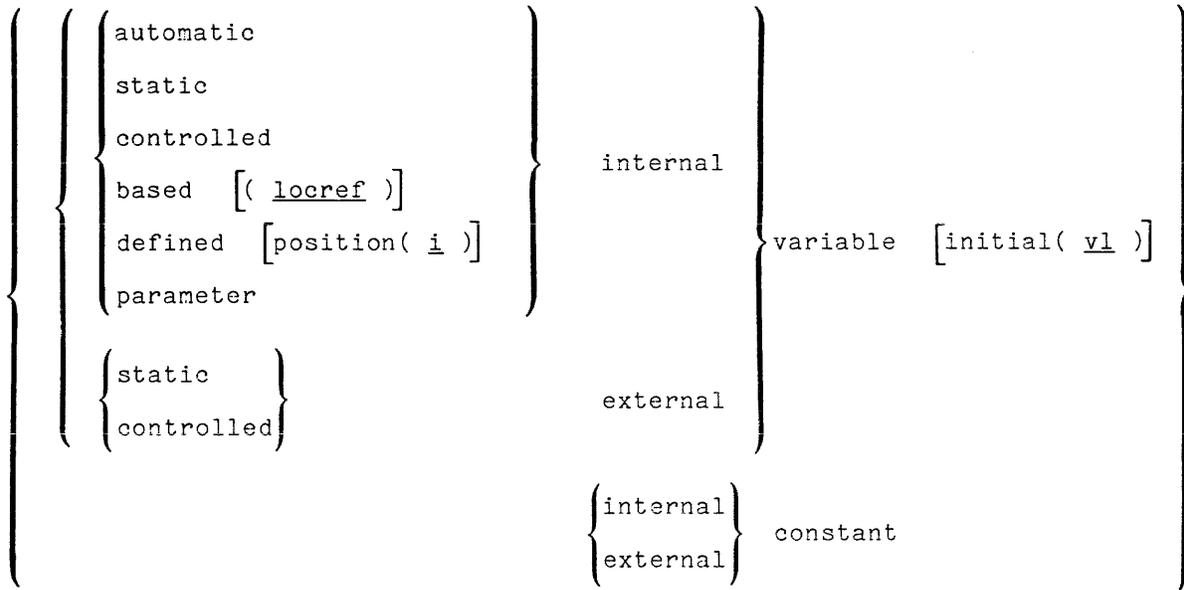
According to Section VI, "Declarations," the declaration of a name is usually divided into two parts. The first part is the storage type, which specifies the kind of values accommodated by the designated variable or constant. The second part is the management class, which specifies various information about the handling of storage including, but not restricted to, the method of allocation. The storage type was defined earlier, in Section III, "Value Storage." The management class is described here.

The management class is composed of four kinds of attributes, as follows:

- The usage category attribute is 'variable' or 'constant'. The attribute describes the way in which the storage is used by the program.
- The scope attribute is either 'internal' or 'external'. The attribute helps determine the region in which the storage is allocated and thus affects the accessibility of the storage.
- The class attribute is 'automatic', 'static', 'controlled', 'based', 'defined', or 'parameter'. The attribute selects the mechanism that invokes the allocation and freeing operations.
- The initial value attribute, when present, specifies a value that is assigned to a variable when it is allocated.

Form of the Management Class

Details of the form of the management class are given by the following diagram:



In this diagram,

- locref is a reference to a locative value or a function that yields a locative value.
- i is an expression whose value can be converted to an integer.
- vl is a list of initial value expressions for the 'initial' attribute and has the syntax given later in this section under "The Initialization Attribute".

The 'initial' attribute must not be used with either the 'defined' or the 'parameter' attribute.

Abbreviations and Defaults

PL/I provides several abbreviations for the management class attributes, as follows:

<u>Attribute</u>	<u>Abbreviation</u>
internal	int
external	ext
automatic	auto
controlled	ctl
parameter	parm
defined	def
initial	init

The default attributes for the category, scope and class are as follows:

<u>Omitted Attribute</u>	<u>Default</u>
<u>category</u>	default is 'variable' exception: default is 'constant' when the attribute 'entry' or 'file' is present in the storage type and no attribute is present that cannot apply to a constant.
<u>scope</u>	default is 'internal' exception: default is 'external' when the attribute 'entry' or 'file' is present in the storage type.
<u>class</u>	default is 'automatic' exception: default is 'static' when both 'external' and 'variable' attributes are present.

The following examples show the application of these defaults and abbreviations to typical attribute sets:

<u>Full Allocation Type</u>	<u>Shortest Form</u>
float automatic internal variable	float
float static external variable	float ext
file external constant	file

DEFAULT RULES

The default rules for the management class each choose the most commonly used attribute as the (nonexceptional) default: most names declared in 'declare' statements are indeed variables rather than constants; an 'internal' name is used wherever possible because it is more economical; and 'automatic' variables are the hallmark of programming in a block-structured language like PL/I.

The exceptions for category and scope are not obvious and require further discussion. They reflect certain special patterns of PL/I programming and are best explained in terms of two specific 'declare' statements:

```
dcl P2 entry external constant;
dcl sysin file external constant;
```

Declarations like these are the most common uses of the 'declare' statement for 'entry' or 'file' names. Therefore, the default rules are adjusted so that these declarations can be given succinctly as

```
dcl P2 entry;
dcl sysin file;
```

The first statement is used when an external procedure (say, 'P1') calls another external procedure at an entry specified by 'P2'. Since 'P2' does not appear in a label prefix in the procedure 'P1', it must be declared in that procedure by a 'declare' statement. The second statement is used to declare the name of a file constant. Something like it must occur in every program that performs input/output.

The exception for class is necessary because there is no 'automatic' class for an 'external' variable. The exception uses 'static' as the default because it is simpler and more economical than 'controlled'.

USAGE CATEGORIES

The usage category attributes indicate the ways in which names are used.

Variables

A variable storage unit sequence, or variable for short, is used to store a value that can change repeatedly throughout its existence and can be set directly by a program statement. A variable can be allocated and freed by the PL/I processor or explicitly by program statements; but in every case, the details of the operation are determined by the scope and class attributes that are given in the declaration of the variable name.

A variable can be declared with any storage type allowed in the language. A variable can accommodate a scalar value, in which case it is a single storage unit, or it can accommodate an aggregate value, in which case it is a sequence of storage units.

Constants

A constant storage unit sequence, or constant for short, is used to store a value that is set by the PL/I processor and does not change. Only the value of a constant can be used in a program; it is not possible to get the address of a constant.

Computational constants are designated by literal constant references, while noncomputational constants are designated by names. These program constructs are described later, in Section VIII, "Expressions."

SCOPES

The scope attribute determines whether or not two declarations of the same name have the same meaning.

Internal Names

Each time a given name is declared in a different block with the 'internal' attribute it has a different meaning. Each declaration can associate a different set of attributes with the name. When storage is allocated for the name, it will be different from that for other declarations of the name because it will be allocated in a region uniquely associated with the block in which the declaration is established.

External Names

Each time a given name is declared in a different block (but in the same process) with the 'external' attribute it has the same meaning. Each declaration must associate exactly the same set of attributes with the name (after defaults and abbreviations have been filled in). Storage is allocated for the name only once, and is shared by all 'external' declarations of the name because it is allocated in a region associated with the program as a whole.

External variables can be allocated outside of the storage that is controlled by the PL/I processor and can exist independently of the execution of a program. Details are given later, in Section XVI, "PL/I in the Multics System," under the heading "External Variables."

Guidelines for the Scope

The 'internal' attribute should be used except where there is a need to share a variable or constant among several external procedures. External storage is much more expensive to allocate and reference than internal storage. The first reference to an external variable can consume as much processor time as the execution of a typical small program.

When external names are used, special care must be taken. If the external procedures of a large program are written at separate times or by different people, the use of external names must be coordinated. Otherwise, conflicting uses of the same external name will be discovered only when the complete program is executed, if at all.

The high cost of external names can be reduced by subjecting the program to the Multics bind command (see the Multics Programmers Manual). This command takes the separately compiled object segments of the external procedures and produces a single object segment equivalent to them. In the resulting object segment, the cost of references to external names may be reduced, particularly if the named variables are bound in with the program. A disadvantage of using the bind command is the extra cost incurred by the bind command itself. It is best applied when a program has been accepted for production use.

Correspondence Between Names and Storage

It is a principle of PL/I that each allocated portion of storage is designated by a name (that is, a declared identifier) in the program. Thus there is a correspondence between names and storage. This is a useful principle; but it does have some exceptions.

An allocated portion of storage is not designated by a name in the following cases:

- Temporary storage units are not designated by names (or any other program constructs). This is because temporaries are accessible to the PL/I processor but not to program statements.
- Computational constants are designated by literal constant references, not names. This is because the spelling of a literal constant is used to indicate the value of the constant. (For example, the literal constant '100' indicates that its value is one hundred.)
- Explicitly allocated based variables are designated by locative values, not names. More is said of this later in this section.

A name does not designate an allocated portion of storage in the following cases:

- A generic function name is used as a kind of macro name, and is replaced by an entry constant name (which does designate storage) before program execution begins.
- A built-in function name has an intrinsic meaning, and, in that respect, resembles an operator such as '+' or '='; for example, when 'abs' is declared 'builtin', it refers to the absolute value operation that is part of the definition of PL/I.
- A condition name is related to an entry variable name; but the storage it uses is not directly accessible to program statements and is managed in a specialized way.
- A constant name is evaluated during program execution.
- A controlled variable name does not designate storage unless it is allocated.

Every name in a PL/I program has a scope. The scopes for the exceptional names just mentioned are 'internal' for generic function references and built-in function names and 'external' for condition names. These exceptional names are discussed in later sections.

STORAGE CLASSES

The storage class attribute determines storage management. By writing a single attribute for a variable name, the programmer selects from six very different mechanisms for storage management. Some mechanisms are easy to learn and use ('automatic', 'static', and 'controlled'). One is designed for the special purpose of transmitting arguments to procedures ('parameter'). Those remaining ('based' and 'defined') are for advanced programming applications and can be ignored in an initial study of PL/I.

Automatic Variables

The attribute 'automatic' designates a storage management mechanism that is driven by the block structure of the program rather than program statements. A variable whose name is declared 'automatic' is allocated as part of the activation of the block in which it is declared and is freed as part of the deactivation of that block. Thus it is available for use during a particular activation of the given block.

The activation of a block occurs as control enters a block (either by invocation of a procedure block or sequential flow into a 'begin' block); it is performed before the first executable statement of the block is executed. The deactivation occurs as control leaves the block (by execution of an 'end' statement, a 'return' statement, or a 'goto' statement); it is performed before execution of a statement that is not contained in the block.

An automatic variable always has 'internal' scope. It is allocated in the activation internal region that is associated with the given activation of the given block.

VARIABLE EXPRESSIONS IN ATTRIBUTES FOR 'automatic' VARIABLES

The attributes of an automatic variable can contain variable expressions; these can be extent expressions (array bounds, maximum string length, or area size) or initial value expressions. When an automatic variable is allocated as part of block activation, these variable expressions must be evaluated. They must depend only on variables that are set before the given block activation begins.

This rule appears to be obvious, but the following program shows that it does introduce a restriction on the use of automatic variables:

```
P1:  proc;
      dcl n fixed bin initial(5);
      dcl A(n) float bin;
      ...
      end;
```

(Since no storage class is given for 'n' and 'A', both are 'automatic'.) Although it is quite clear what the programmer wants to have happen, this program is invalid because the allocation of 'A' depends on a variable, 'n', that was not set before block activation began. In contrast, consider the program:

```
P2:  proc;
      dcl n fixed bin static initial(5);
      dcl A(n) float bin;
      ...
      end;
```

This program is valid because a 'static' variable is allocated and set before block activation begins.

SAVING VARIABLE EXTENTS FOR 'automatic' VARIABLES

The storage type used for every reference to a variable must be consistent; that is, it must be identical to the storage type used for the allocation of that variable. This is a fundamental rule of PL/I; it asserts that storage cannot be given more than one interpretation. There are exceptions to this rule, but they apply only when the storage class is 'based' or 'defined', and are relevant only in advanced applications of the language.

When 'based' or 'defined' variables are not used, the design of PL/I makes it impossible for the programmer to break the rule of storage type consistency; specifically, when storage is allocated for a variable, every part of the storage type that can change during program execution is saved. Each reference to the variable uses the saved information and is therefore consistent with the allocation of the variable.

The following program fragment shows the effect of this treatment of storage types:

```
P1:  proc;
      dcl n fixed bin;
      n = 4;
      ...
      begin;
      dcl S char(n+2);
      ...
      n = 100;
      S = "abcdef";
      ...
      end;
      ...
      end;
```

(Since no storage class is given for 'n' and 'S', both of the designated variables are 'automatic'.) At the time 'S' is allocated, the storage type given by the 'declare' statement is 'char(6)'; this storage type is used for the allocation and is saved. Subsequent reference to 'S', including both the assignment to 'S' and the automatic freeing of 'S' at the deactivation of the 'begin' block, use the saved storage type. Thus the fact that the storage type given by the 'declare' statement changes to become 'char(102)' has no effect on the program and, specifically, does not cause a violation of the storage type consistency rule.

The extents for an 'automatic' variable are saved in temporaries that are allocated and freed in the same region and at the same time as the variable. Because temporary storage is not accessible to program statements, the saved extents cannot be inadvertently changed.

Static Variables

The attribute 'static' designates a storage management mechanism that makes the allocated storage available throughout program execution. A variable whose name is declared 'static' is allocated at some time before the first reference to the variable is processed and is freed at some time after the last reference is processed.

An ordinary static variable can have 'internal' or 'external' scope and is allocated in a permanent internal region or in the ordinary external region, accordingly.

VARIABLE EXPRESSIONS IN ATTRIBUTES FOR 'static' VARIABLES

Every expression in an attribute in the declaration of a 'static' variable must be a constant expression. In particular, the extents (array bounds, maximum string length, or area size) and the initialization expressions must be constants. Since the extents are constant, it is not necessary to save them.

Controlled Variables

The attribute 'controlled' designates a storage management mechanism that is driven by program statements and is thus under program control. A variable whose name is declared 'controlled' is allocated by the execution of an 'allocate' statement and freed by the execution of a 'free' statement. Thus the variable is available for whatever portion of execution of the program the programmer requires. A small control block permanently associated with the 'controlled' variable is used to locate its currently allocated storage.

A controlled variable can have 'internal' or 'external' scope and its control block is allocated in a permanent internal region or in the ordinary external region, accordingly.

THE 'allocate' AND 'free' STATEMENTS FOR 'controlled' VARIABLES

A controlled variable whose name is id is allocated by the statement

```
allocate id;
```

and is freed by the statement

```
free id;
```

The keyword 'allocate' can be abbreviated as 'alloc'. More than one identifier can be mentioned in an 'allocate' or 'free' statement; in that case, each identifier is separated from the next by a comma.

STACKING CONTROLLED VARIABLES

A simple use of a controlled variable name is to allocate storage for a variable, reference the storage as often as necessary, and then free the storage. A more general use of a controlled variable name is to allocate storage for a variable more than once before freeing its storage. When used in this way, a controlled variable name designates a stack of variables, and has the following properties:

- Each time the name is used in an 'allocate' statement, a new variable is allocated and earlier allocations remain undisturbed.
- Each time the name is used in a 'free' statement, the most recently allocated variable designated by the name is freed and the less recent allocations are undisturbed.

- Each time the name is used as a reference, the most recently allocated variable designated by the name is accessed for the setting or retrieving of a value.

A program that uses a name in a reference or a freeing when there is no variable allocated for that name is invalid. This can occur when no allocation has occurred or when the number of freeings already equals the number of allocations.

As an example of the use of a controlled variable name to designate a stack of variables, consider the following program:

```
P1:  proc;
      decl x float bin controlled;
      ...(Computation 1)
      allocate x;
      x = 10;
      ...(Computation 2)
      allocate x;
      x = 20;
      ...(Computation 3)
      free x;
      ...(Computation 4)
      free x;
      ...(Computation 5)
      end;
```

The "Computations" contain only references to 'x' that retrieve its values; all allocations, freeings, and setting are shown explicitly. A reference to 'x' in Computation 1 would be invalid because no variable has been allocated for 'x'. References in Computations 2 and 3 yield 10 and 20. A reference in Computation 4 yields 10 because the variable first allocated for 'x' becomes available after the first 'free' statement; this is the point of the example. A reference to 'x' in Computation 5 is invalid because all variables associated with 'x' have been freed.

VARIABLE EXPRESSIONS IN ATTRIBUTES FOR 'controlled' VARIABLES

The execution of an 'allocate' statement causes the extent expressions (array bounds, maximum string length, or area size) and the initial value expressions in the declaration of the allocated variable name to be evaluated. The expressions can depend on any variable that has a value immediately before execution of the 'allocate' statement.

Sometimes a variable expression in an attribute has access to different variables than the 'allocate' statement that causes evaluation of the expression. Consider the program fragment:

```
P1:  proc;
      decl n fixed bin;
      decl A(n+2) float bin controlled;
      n = 1;
      ...
      begin;
      decl n fixed bin;
      n = 10;
      ...
      allocate A;
      ...
      end;
      ...
      end;
```

The 'allocate' statement uses the storage type 'dim(3) float' for the allocation of 'A', not 'dim(12) float'. The example emphasizes the rule, true throughout PL/I, that the resolution of a name depends on where it occurs in the program, not what it is used for.

SAVING VARIABLE EXTENTS FOR 'controlled' VARIABLES

The values of any variable extents in the storage type of a 'controlled' variable name are saved, as with 'automatic' variable names. The value saved and used for subsequent references is that computed when the variable is allocated; it is saved in a temporary that is associated with the variable. When more than one variable is allocated for a given controlled variable name, the saved extents may not be the same; they are evaluated for each allocation of the variable.

Parameter Variables

The attribute 'parameter' designates a storage management scheme that is an integral part of the invocation of a procedure, by either a 'call' statement or a function reference. A variable name declared 'parameter' never has storage allocated for it; instead, it is associated with storage that has previously been allocated and that contains an argument for the procedure invocation. The parameter name is associated with the argument variable as part of the activation of the procedure block, and the association is broken as part of the deactivation of the block.

A parameter variable name always has 'internal' scope. Because a parameter variable name is associated with a variable allocated for some other name, the storage it references can be in any region.

VARIABLE EXPRESSIONS IN ATTRIBUTES FOR 'parameter' VARIABLES

Each extent expression (array bound, maximum string length, or area size) must be either a constant expression or a single asterisk, '*'. When an extent expression is an asterisk, the corresponding extent of the associated argument variable is used. Full details of this mechanism, which is especially provided for parameters, is given later in Section XII, "Procedure Invocation."

A parameter variable never has storage allocated for it; therefore, it cannot have an initial attribute and the treatment of variable expressions in that attribute does not arise.

Based Variables

The attribute 'based' designates a storage management mechanism that can be driven by program statements. A variable whose name is declared 'based' can be allocated by an 'allocate' statement and freed by a 'free' statement. Such a variable is called an explicitly allocated based variable.

A based variable always has 'internal' scope. An explicitly allocated based variable is allocated either in a permanent internal region or in an area variable; the choice depends on the 'allocate' statement.

Whenever a variable is allocated for a based variable name, the storage type is supplied by the name but the name is not associated with the variable as its designator. Instead, the allocation operation produces a locative value that is assigned to a 'pointer' or 'offset' variable.

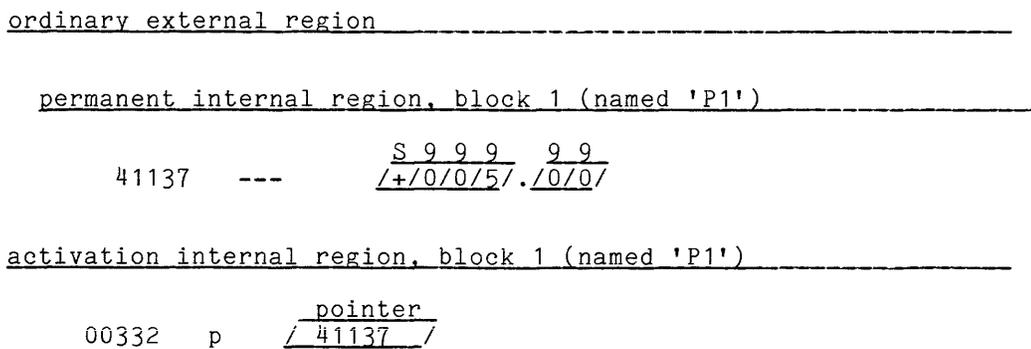
Whenever a variable is referenced with a based variable name, the storage type is supplied by the based variable name, but the name does not locate the variable in storage. Instead, the variable is designated by a locative value that is obtained by a 'pointer' or 'offset' variable that is mentioned with the based variable in the reference.

A simple example of the use of a based variable and an associated pointer is:

```
P1: proc;
    dcl A dec(5,2) based initial(0);
    dcl p pointer;
    dcl sysprint file;
    ...
    allocate A set(p);
    ...
    p->A = 5;
    ...
    put list (p->A);
    ...
    free p->A;
end;
```

In this example, the based variable name 'A' is used in the allocation of an explicitly allocated based variable; but it is the value of the pointer 'p' that designates that variable, not the based variable name 'A'. Wherever 'A' is used, the pointer variable name 'p' is used with it.

As the 'put' statement is about to be executed, the diagram for storage is:



Observe that the 'dec(5,2)' variable is not given the name 'A'; instead, it must be designated by its locative value, represented as '41137'.

The purpose of the based variable is not as self-evident as that of the other classes of variable. The discussion that follows gives the rules that govern the use of based variable, and illustrates those rules.

REFERENCE IN 'based' ATTRIBUTE

The 'based' attribute can be followed by a parenthesized reference to a locative value. This reference is adopted as the default locative reference for the variable name to which the attribute applies. If a locator qualifier is omitted from a reference to the based variable, the parenthesized reference is used as an assumed locative qualifier. Similarly, if the 'set' option is omitted from an 'allocate' statement for the based variable, the parenthesized reference is used as an assumed 'set' option.

The example of a based variable given earlier can be shortened by the use of a parenthesized reference with the 'based' attribute. The revised program is as follows:

```
P2:  proc;
      dcl A fixed dec(5,2) based(p) initial(0);
      dcl p pointer;
      dcl sysprint file;
      ...
      allocate A;
      ...
      A = 5;
      ...
      put list(A);
      ...
      free A;
      end;
```

Thus all mentions of the pointer 'p' outside of the declarations are eliminated, and the usage for 'A' closely resembles that of a controlled variable.

'allocate' AND 'free' STATEMENTS FOR 'based' VARIABLES

A based variable whose name is id is allocated by a statement of the form

```
allocate id [set( locref )] [in arearef ] ;
```

and is freed by a statement of the form

```
free id [in( arearef )] ;
```

The square brackets indicate that the options can be omitted. The locref in the 'set' option must be a reference to which a locative value can be assigned, namely a 'pointer' or 'offset' variable. The arearef in the 'in' option must be a reference that designates an 'area' variable.

If id has been declared 'based(ref)', then the 'set' option can be omitted and the PL/I processor assumes 'set(ref)'. If the ref supplied by the 'based' attribute is not a reference to which a locative value can be assigned, however, the assumed option will be invalid. When the 'allocate' statement is executed, the locative value that designates the allocated storage is assigned to the variable supplied by the 'set' option or the 'based' attribute.

If a programmer wants to allocate id in the internal permanent region, then the 'in' option is omitted. If a programmer wants to allocate id in some 'area' variable (which can be in any region), then the area is specified by means of the 'in' option. When id is freed, the 'in' option must be the same that was used for its allocation.

VARIABLE EXPRESSIONS IN ATTRIBUTES

The execution of an 'allocate' statement causes the extent expressions (array bounds, maximum string length, or area size) and the initial value expressions in the declaration of the allocated variable name to be evaluated. The expressions can depend on any variable that has a value immediately before execution of the 'allocate' statement. In this respect, a based variable name behaves exactly as a controlled variable name.

SAVING OF VARIABLE EXTENTS

The values of variable extents in the storage type of a based variable name are not saved automatically by the PL/I processor; in this respect, based variables differ from all others. The extents in the storage type are evaluated for each reference; and in these cases any change in an extent results in violation of the storage type consistency rule and an invalid program.

The following example is similar to a valid program fragment given earlier, in the description of automatic variables:

```
P1:  proc;
      dcl n fixed bin;
      dcl S char(n+2) based(beta);
      dcl beta pointer;
      ...
      n = 4;
      allocate S;
      ...
      n = 100;
      S = "abcdef";
      ...
      free S;
      end;
```

This program is not valid. The reference to 'S' in the assignment statement causes the extent expression 'n+2' to be evaluated (since there is no stored extent available). Thus, although the variable was allocated with storage type 'char(6)', it is referenced with storage type 'char(102)'. The string "abcdef" is filled out with blanks to make a string of 102 characters, and that string is assigned to 'S', where storage for only six characters has been provided. The result is the overwriting of variables allocated immediately after 'S'. A similar error occurs when the storage for 'S' is freed. These errors are not detected by the PL/I processor, and their effect may be an obscure malfunction of the program.

The invalid program just given can be repaired by keeping the expression for the maximum length of 'S' unchanged throughout the existence of the variable for 'S'. Toward this end, a new variable, 'n2' can be declared and used in the extent expression while the value of 'n' is allowed to change as before. The revised program is:

```
P2:  proc;
      dcl n fixed bin;
      dcl n2 fixed bin;
      dcl S char(n2+2) based(beta);
      dcl beta pointer;
      ...
      n = 4;
      n2 = n;
      allocate S;
      ...
      n = 100;
      S = "abcdef";
      ...
      free S;
      end;
```

This program is valid. It will evaluate the extent of the 'char' attribute three times (for the allocation, the reference, and the freeing), but it will obtain the value '6' each time.

'refer' OPTION

PL/I has a special feature, the 'refer' option, that makes possible the systematic handling of variable extents of based variables. Using the 'refer' option, the program fragment considered earlier can be written as follows:

```
P3:  proc;
      dcl n fixed bin;
      dcl 01 Spair based(beta),
           02 n2 fixed,
           02 S char(n+2 refer(n2));
      dcl beta pointer;
      ...
      n = 4;
      allocate Spair;
      ...
      n = 100;
      S = "abcdef";
      ...
      free Spair;
      end;
```

This revision of the original, invalid program has changed only the handling of 'S', replacing it with the structure 'Spair'; however, the change eliminates the storage type inconsistency. The extent expression 'n+2 refer(n2)' is interpreted as follows:

When 'Spair' is allocated, calculate the extent for 'Spair.S' (also known as 'S') from the expression 'n+2'; use that value for the allocation of storage; and then save the value in 'Spair.n2' (also known as 'n2'). Furthermore, whenever a reference to 'S' is made, refer to 'n2', not the expression 'n+2', for the value of the extent.

Evidently, the 'refer' option has an important effect on the interpretation of the program.

The 'refer' option is used to save the variable extents of a based variable name in much the same way that the PL/I processor automatically saves variable extents for all other storage classes. Any number of extents in a variable can each be handled by a refer option. For example, a structure that would otherwise be declared as

```
dcl 01 alpha(-2:r-2) based,  
    02 beta bit(J(m-1)) varying,  
    02 gamma(s1,s2) float bin;
```

can be declared as

```
dcl 01 X based,  
    02 (k1,k2,k3,k4) fixed bin,  
    02 alpha(-2:r-2 refer(k1)),  
    03 beta bit(J(m-1) refer(k2)) varying,  
    03 gamma(s1 refer(k3),s2 refer(k4)) float;
```

Such a variable is called a self-defining structure because it carries its own extents within itself.

The parenthesized reference that follows the 'refer' keyword must designate a scalar variable that is declared earlier in the same structure that contains the 'refer' option. The referenced variable must, of course, be of suitable data type to have the extent value assigned to it.

EQUIVALENCED BASED VARIABLES

All of the based variables thus far discussed have been explicitly allocated based variables; that is, they are variables that were allocated using a based variable name to supply the storage type; and the locative value that designates the variable was produced as part of the allocation operation.

A second kind of based variable is the equivalenced based variable. Such a variable does not have storage of its own, but rather is superimposed on or equivalenced to a variable, the base variable, that was previously allocated. The base variable can have any storage type; it can even be an explicitly allocated based variable. However, the base variable must be connected; that is, if it is an aggregate, then its components must occupy an uninterrupted sequence of storage units.

The locative value that designates the base variable is obtained by the 'addr' built-in function; for example, the assignment statement

```
p = addr(A);
```

obtains the locative value that designates the variable designated by 'A' and assigns that locative value to the pointer variable designated by 'p'.

There are three kinds of equivalenced based variables: simple based variables, string overlay based variables, and partial based variables; each of these is discussed in the following paragraphs.

Simple Based Variables

The storage type of a simple based variable must agree exactly with the storage type of the base variable.

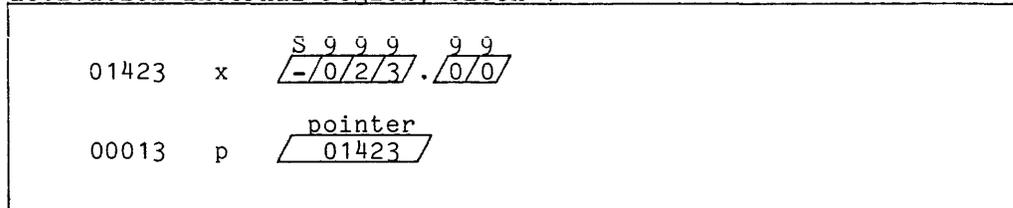
As an example of the use of a simple based variable, consider the following program:

```
P1:  proc;
      dcl x fixed dec(5,2);
      dcl y fixed dec(5,2) based;
      dcl p pointer;
      dcl (sysin,sysprint) file;
      p = addr(x);
      get list(x);
      put list(2*p->y);
      end;
```

This program inputs a number, doubles it, and prints it. The program would be invalid if the storage type of 'y' differed from that of 'x'.

Immediately after the 'get' statement has input the value '-23' from the input stream, the diagram for storage is:

activation internal region, block 1



As is always the case with a based variable, there is no mention of 'y' in storage; it is used only to supply a storage type for use in conjunction with 'p'.

As a second example of a simple based variable, consider the following statements:

```
dcl 01 A(5),
      02 X fixed bin,
      02 Y char(6);
dcl 01 B based,
      02 R fixed bin,
      02 S char(6);
dcl p2 pointer;
```

Suppose that the following assignment statement is executed:

```
p2 = addr(A(3));
```

Then the following references are valid:

```
p2->B      (means 'A(3)')
p2->B.R    (means 'A(3).X')
p2->B.S    (means 'A(3).Y')
```

Thus the 'pointer' variable supplies the address of the leftmost name, 'B' in this example, in a reference to a based variable. Further examples are given later, in Section VIII, "Expressions," under "Locator-Qualified Variable References".

String Overlay Based Variables

A string is not a truly indivisible value; it is made up of characters or bits. In some applications, it is useful to impose more than one interpretation on the storage for a sequence of characters or bits. String overlay based variables are used for this purpose.

A string overlay based variable and its base variable must be either 'nonvarying unaligned' scalars or aggregates of 'nonvarying unaligned' scalars. For a given string overlay, all variables must be 'character' or all variables must be 'bit'. The aggregate type of the base variable need not match, and a pictured character-string variable can be matched with an ordinary character-string variable. This relaxation of the rules for aggregate type and pictured data type permits different interpretations of a given string variable.

As an example of a string overlay based variable, consider the following statements:

```
dcl word(1024) bit(36);
dcl 01 float_num based,
    02 sign bit(1) unal,
    02 characteristic bit(8) unal,
    02 mantissa bit(27) unal;
dcl pnum pointer;
```

Suppose the following assignment statement is executed:

```
pnum = addr(word(23));
```

then the following references are valid:

```
pnum->float_num      (means 'word(23)')
pnum->float_num.sign  (means the first bit of 'word(23)')
pnum->float_num.characteristic (means the next 8 bits of 'word(23)')
pnum->float_num.mantissa    (means the last 27 bits of 'word(23)')
```

Thus the based structure variable can be used to designate various substrings of the string variable designated by 'word(23)'.

The requirement that strings be 'nonvarying unaligned' is the way of saying, in PL/I, that several strings must be stored together, without any unused storage or any word or bit counts between them. Because the example satisfies this restriction, a single bit string can be interpreted as a structure made up of three bit strings.

Partial Based Variables

Sometimes it is necessary to examine a portion of a structure variable without knowing all the details of the storage type of the variable. A partial based variable is used for this purpose.

A partial based variable and its base variable must be structure variables whose storage types agree through the first n declaration clauses. For example, consider the following statements:

```
dcl 01 alpha,
    02 A fixed bin,
    02 B,
        03 B1 float bin,
        03 B2 char(10),
    02 C char(2000) varying;
dcl 01 beta based,
    02 Q fixed bin,
    02 R,
        03 R1 float bin,
        03 R2 char(12),
    02 S char(2000) varying;
```

Observe that the first four declaration clauses of the storage types of 'alpha' and 'beta' are the same; that is, both storage types begin:

```
01, 02 fixed, 02, 03 float
```

Thus 'beta' can be used as a partial based variable name with 'alpha' as the base variable name.

There are two restrictions on references to a partial based variable, as follows:

- When the storage types of the based variable name and the base variable name are compared from the beginning (the level-one declaration clause) toward the end (the last declaration clause), the storage types must agree through the declaration of all components that are designated by the given reference.
- Furthermore, if one of the components designated by the reference is a component of a level-two structure, then the declarations of the based variable and the base variable must agree for all components of that level-two structure.

Consider again the declarations of 'alpha' and 'beta' given earlier in this discussion. Suppose the following assignment statement has been executed:

```
p = addr(alpha);
```

where 'p' is declared 'pointer'. In this situation, the only valid use of 'beta' is the following reference:

```
p->beta.Q      (means 'alpha.A')
```

Thus it is possible to obtain the value of 'alpha.A' without matching the whole storage type of 'alpha'.

Since the storage types of 'alpha' and 'beta' agree through 'alpha.B.B1', it might seem valid to use 'beta' in the following reference:

```
p->beta.R.R1
```

However, this reference violates the second restriction on partial based references. Specifically, the reference designates a component 'alpha.B.B1' of a level-two structure 'alpha.B' that has a component 'alpha.B.B2' for which the storage type of 'alpha' and 'beta' do not agree.

One use for a partial based variable reference arises in handling input from a file in which records have several different structures. If the file is designed so that every record begins with an integer value that indicates the structure of the remainder of the record, then this integer can be obtained by means of a partial based variable reference. An example is given later, in Section XV, "Record Input/Output" under "An Example of Based Input".

Defined Variables

The attribute 'defined' designates a storage management mechanism that provides a new name for an existing variable rather than allocating a new variable. The association between the defined variable name and the designated variable is formed at the time of block activation and broken at block deactivation.

A defined variable name always has 'internal' scope. Because a defined name is associated with a variable allocated for some other name, the storage it references can be in any region.

VARIABLE EXPRESSIONS IN ATTRIBUTES FOR 'defined' VARIABLES

The extent expressions (array bounds, maximum string length, or area size) are evaluated as part of the activation of the block in which the defined variable is declared. Therefore, the expressions can depend only on variables that are defined before block activation begins.

A defined variable never has storage allocated for it; therefore, it cannot have an initial attribute and the treatment of variable expressions in that attribute does not arise.

SAVING VARIABLE EXTENTS FOR 'defined' VARIABLES

The values of the extent expression are saved in temporaries at the time they are computed. The storage for these temporaries is allocated in the activation region associated with the block activation, and the storage is freed upon block activation. Because the extents are saved, it is not possible to violate the storage consistency rule in the use of a defined variable.

USES OF DEFINED VARIABLES

The 'defined' attribute consists of the keyword 'defined' followed by a parenthesized reference. The variable designated by the reference is called the base variable because it is the base on which the defined variable is superimposed. There are three ways to use a defined variable: simple defining, string overlay defining and isub defining.

Simple Defining

For most cases of simple defining, the base variable must have the same storage type as the defined variable. Some examples follow; consider first:

```
dcl S char(20) varying static;
dcl T char(20) varying defined(S);
```

In this case, any reference to 'T' will be equivalent to a reference to 'S'. Consider next:

```
dcl 01 A,
    02 B(n),
    03 C float bin,
    03 D float bin,
    02 E char(6);
dcl X float defined(A.B(i-2).D);
dcl Y(n) float defined(A.B(*).D);
dcl 01 Z defined(A.B(j)),
    02 Z1 float bin,
    02 Z2 float bin;
```

In this case, 'X' is associated with the scalar designated by 'A.B(i-2).D'; the variable 'i' is resolved where it appears (in the declaration) but is evaluated each time a reference to 'X' is processed. The defined variable 'Y' is associated with a cross-section of the array 'A.B', namely that formed of the second member of each element of the array. The defined variable 'Z' is associated with a particular element of 'A.B'; once again, 'j' is evaluated for each reference.

String Overlay Defining

String overlay defining allows a string variable to be defined onto the storage of another string variable so that the defined variable occupies all or part of the same storage allocated for the base variable. String overlay defining can be performed for either bit strings or character strings, but the two types cannot be mixed in any given use of string overlay defining.

The defined variable and the base variable must be 'nonvarying unaligned' string scalars or aggregates of 'nonvarying unaligned' string scalars. The aggregate type of the defined variable and the base variable need not match, and a pictured character string variable can be treated as an ordinary 'nonvarying' character string variable; these are the "loopholes" that make string overlay defining useful.

Examples of string overlay defining are given in the following 'declare' statements:

```
dcl A(5) char(2) unal;
dcl B char(8) def(A);
dcl 01 C def(A),
      02 X char(5) unal,
      02 Y char(5) unal;
```

The first statement declares 'A' as the name of an array variable that has five elements, each composed of two characters. The second statement declares 'B' as another name for the variable designated by 'A'; but it views the variable not as an array but as a sequence of ten character positions, the first eight of which are interpreted as a scalar character-string variable. The third statement declares 'C' as yet another name for the variable designated by 'A'; but it interprets the character positions of 'A' as a structure composed of two five-character members.

The 'position' attribute can be used to start the 'defined' variable at some position other than the first character position of the based variable. For example, consider:

```
dcl D char(5) def(A) pos(6);
```

This statement declares a name for the last five character positions of the variable designated by 'A'.

Sub Defining

For sub defining, the special lexemes '1sub', '2sub', '3sub', and so on, are used to make special use of the subscript values of the defined variable. Consider the declarations:

```
dcl A(3,3) float bin;
dcl Q(3) float bin defined A(1sub,1sub);
dcl R(3,3) float bin defined A(4-1sub,2sub);
```

In this case, 'Q' is made up of the three elements of 'A' that lie on the diagonal, and 'R' is made up of the rows of 'A' in reverse order. The two arrays map onto 'A' as follows:

Q(1) -- A(1,1)
Q(2) -- A(2,2)
Q(3) -- A(3,3)

R(1,1) -- A(3,1) R(1,2) -- A(3,2) R(1,3) -- A(3,3)
R(2,1) -- A(2,1) R(2,2) -- A(2,2) R(2,3) -- A(2,3)
R(3,1) -- A(1,1) R(3,2) -- A(1,2) R(3,3) -- A(1,3)

Guidelines for the Storage Class

Suggested uses for each of the six storage classes are given here.

AUTOMATIC VARIABLES

An automatic variable should be used wherever practical; the fact that 'automatic' is the default storage class attribute makes this recommendation easy to follow. An automatic variable is clearly associated with a block and is allocated only while that block is activated; therefore, its intended role in a program is more clear than that of any other class of variable. When a block is considered as a building block of a larger program, the automatic variable names declared in the block can be entirely ignored; their existence cannot be perceived from outside the block.

A reference to an automatic variable that is immediately contained in the block in which the variable name is declared is a local reference. Local references are considerably less costly to process than other references. In order to maximize the number of local references, an automatic variable should always be declared in the smallest existing block that contains all references to it; if this rule is followed, at least some of the references to the variable are local references.

If the programmer allocated a particular variable (controlled or based) at the beginning of execution of each block and freed it at the end of the block, the storage management operations would incur a considerable expense. Although an automatic variable is allocated and freed for each execution of its block, an extra cost is not incurred; on the contrary, the special techniques used in the implementation of PL/I (the use of a stack) make the cost of storage management of an automatic variable negligible.

The use of variable expressions in the extents of a storage type of an automatic variable increases the cost of the variable. Of course the extent expressions must be evaluated and saved when the variable is allocated; but this cost is obvious and is examined like any other computation cost. A more important cost arises in certain cases in references to the variable. Consider the declaration:

```
dcl 01 A,  
    02 B(n) float bin,  
    02 C fixed bin;
```

For each reference to 'C', the PL/I processor must not only find the location at which the structure 'A' begins, but also must determine the size of 'B' so it can be skipped over to reach 'C'. The size of 'B' is not known at compile time, and therefore relatively inefficient code for the reference must be compiled. A more efficient declaration of the same structure is:

```
dcl 01 A,  
    02 C fixed bin,  
    02 B(n) float bin;
```

When other considerations do not forbid, components with variable extents should be placed as late in a structure as possible.

STATIC VARIABLES

A static variable is used when a variable must exist outside the activation of the block in which the variable is declared. This requirement can arise in the following ways:

- A variable that is shared between several external procedures (compilable units) must be 'external' and must therefore be 'static' or 'controlled'. An external 'static' variable is less costly than an external 'controlled' variable and therefore should be used where possible. Once again, the default conventions make this recommendation easy to follow.
- When a variable is used to keep information throughout the various invocations of a procedure, an 'internal static' variable should be used. Such a variable can be used, for example, to count the number of times a procedure is invoked in a process.

CONTROLLED VARIABLES

A controlled variable is used when the built-in storage management mechanism of the automatic or static storage class does not suit a particular application. This requirement can arise in the following ways:

- When a stack of variables is needed, a controlled variable is a convenient choice. When a program is modified to be reentrant the static variables can be replaced by controlled variables; then each variable is allocated (pushed down) before each reentry and freed (popped up) after the reentry.
- When an external variable must have variable extents, a controlled external variable is a possible choice.

- When storage is a critical resource, controlled variables can be used to program a minimum use of storage.

PARAMETER VARIABLES

A parameter variable is used for the parameters of a procedure; there is no choice about that. Sometimes, however, the programmer does have a choice between transmitting a value to a procedure by a parameter or by a variable in a block that contains the procedure. It is usually best to use a parameter. A parameter that is by-reference can efficiently deliver a variable, whether it is a scalar or a large aggregate; and it makes the procedure independent of its environment.

BASED VARIABLES

A based variable is often used in a program that uses linked data structures. In such a program, locative values are used as the links and based variables are used for the structures connected by the links. The Multics system itself and the Multics PL/I compiler, both of which are written in PL/I, make extensive use of linked data structures and thus of based variables.

Only based variables can be allocated in an area variable; therefore based variables must be used in order to take advantage of the offset values and other special features that are available with area variables.

Based variables are also used in certain rather special ways. In some applications, it is necessary to operate on an aggregate value without fully knowing its storage type; in these cases, a based variable is used to examine part of the value. In string-processing applications that often arise in business programming, it is useful to interpret a given sequence of characters in more than one way by superimposing different aggregates of string variables; in these cases, based variables are used. These applications represent the situations in which PL/I makes gains in efficiency at the cost of breaking its rule of storage type consistency.

DEFINED VARIABLES

A defined variable is used to associate a new name with an existing variable or part of an existing variable. In certain cases, as when isubs are used with a defined array, the new name can be mapped onto the existing variable in a special way. There are occasions on which a defined variable can do what a based variable can do; and on these occasions the defined variable is usually preferred because defined variables are easier to use and understand.

In practice, defined variables are not often used in Multics PL/I. They are in competition with based variables, and the based variables are considerably more general. They are also in competition with the paging mechanism of PL/I, which greatly reduces the need for programming techniques that save storage. In Multics PL/I programming, a defined variable should seldom be introduced for the sole purpose of using the same storage for two purposes and thus saving storage.

'initial' ATTRIBUTE

The 'initial' attribute is used with the name of a scalar or array variable in order to set the value of that variable when the variable is allocated.

Initialization Syntax

The syntax of the 'initial' attribute is given by the following diagram:

$$\left. \begin{array}{l} \text{initial} \\ \text{init} \end{array} \right\} (\text{ivi} , \dots)$$

where ivi (initial value item) is defined as:

$$\left\{ \begin{array}{l} [(\text{rep})] \left\{ \begin{array}{l} \left\{ \begin{array}{l} + \\ - \\ \wedge \end{array} \right\} \left\{ \begin{array}{l} \text{ref} \\ \text{const} \end{array} \right\} \\ (\text{exp}) \\ * \end{array} \right\} \\ (\text{rep}) (\text{ivi} , \dots) \end{array} \right\}$$

and where:

- rep (replicator) is any expression whose value can be converted to an integer.
- ref is any variable reference or function reference.
- const is a literal constant or is constant complex expression (a real, a sign, and an imaginary).
- exp is any expression.
- ivi,... is a sequence of one or more initial value items separated by commas.

According to the syntax, there are two forms for an initial value item. The first form is the one that provides a single initial value or, if a replicator is used, a sequence of identical values. The syntax makes a special provision for references and constants in order to exempt them from being enclosed in parentheses. Consider the following attribute:

```
initial(-2.8-15i,(n-1)0)
```

The first item specifies a single initial value; the second item specifies a sequence of zero values whose length depends on the value of 'n' when storage is allocated for the variable to which the attribute applies. Consider next the attribute:

```
initial(x,-v(i+2,j),(4)F(a-sin(theta,z)))
```

This attribute could be used to initialize a six-element array (although such variety is unlikely for the initialization of a single array). The first item is a variable name, the second is a subscripted variable reference, and the third is a function reference that is evaluated and has four copies of its value entered in the list.

The use of an asterisk, '*', as an initial value causes the corresponding value to be left undefined; that is, it skips initialization. Consider the following:

```
initial((5)(1,*))
```

If this attribute is applied to an array with ten elements, it will initialize the odd-numbered elements to '1' and leave the others undefined.

The second form of initial value item allows the use of a parenthesized list of items as a sublist of a larger list. In this form, the replicator is required because the only reason for forming a sublist is to replicate it as a whole. Consider, for example, the attribute:

```
initial((3)(-1,(2)0))
```

After application of the first replicator, this attribute is equivalent to:

```
initial(-1,(2)0,-1,(2)0,-1,(2)0)
```

After expansion of the remaining replicators, this attribute is equivalent to:

```
initial(-1,0,0,-1,0,0,-1,0,0)
```

When the replicator is a variable expression, it cannot be expanded until its values are actually required; for example:

```
initial((n)(-1,(m+3)0))
```

depends on the values of 'n' and 'm'.

Use of 'initial' Attribute

An 'initial' attribute cannot be used with a structure name. It follows that the attribute can only be used with a variable name that designates a scalar or an array of scalars. When the attribute is used with a scalar, it must supply exactly one value; when it is used with an array, it must provide one value for each element of the array. Each value must be suitable for assignment to the variable it initializes.

As an example of several uses of the 'initial' attribute, consider the following declaration:

```
dcl 01 A,  
    02 B(m,n-2) float init((m)(1,(n-3)*),  
    02 C char(3) init("xxx"),  
    02 D,  
    03 E float init(0),  
    03 F pointer init(null()),  
    02 G float;
```

In this structure, the array 'A.B' has its first column initialized to '1' and its remaining columns left undefined; the scalar variable 'C', 'E', and 'F' are initialized to appropriate scalar values; and the scalar variable 'G' is not initialized.

An 'initial' attribute is processed only as part of the storage allocation operation. Since a variable name of storage class 'parameter' or 'defined' is never subjected to the storage allocation (but rather is associated with existing storage) it is incorrect to use an 'initial' attribute with such names. A variable name of storage class 'based' can be used either to allocate storage, in which case its 'initial' attribute is processed, or to be associated with existing storage, in which case the 'initial' attribute is ignored.

In the definition of string literal constants, given later in Section VIII, "Expressions," a replicator is allowed for a string constant. For example:

```
(3)"ab"
```

is equivalent to "ababab". The PL/I processor will interpret an initial value replicator as a string replicator if it occurs in an appropriate context. For example, the declaration

```
dcl S(3) char(2) init((3)"ab");
```

is invalid because the initial attribute is expanded to give the wrong result:

```
dcl S(3) char(2) init("ababab");
```

To obtain the desired result, the programmer must write the string replicator '(1)' and then precede that with an initial value replicator:

```
dcl S(3) init((3)(1)"ab");
```

This problem is a flaw in the design of PL/I; it arises only in the initialization of string variables, and the incorrect usage illustrated above is detected by the compiler.

'options(constant)' Attribute

Multics PL/I assigns storage for internal static, initialed variables in either the text section or static (linkage) section of the object segment, depending on whether the variable is only referenced or referenced and set (changed by assignment). Variables that are never set are allocated in the text section for efficiency; the text section is pure, and so all users of the program can share it. If the variable were placed into the impure static section, then each user would get his own copy, and more storage would be used. Occasionally, an internal static, initialed variable may be passed by reference to a function or subroutine. Since Multics PL/I cannot determine whether an argument is input, output, or both, it is forced to assume that every argument passed by reference can be changed or set. Some Multics system subroutines are defined to take only input arguments, however. An example is the `ioa_` subroutine.

The 'options(constant)' attribute has been added to PL/I to force allocation of internal static, initialed values in the text section, even if they are passed by reference. It is up to the programmer to ensure that no subroutine or function attempts to set such a variable. Any attempt would most likely fail, since Multics PL/I removes write permission from the object segment after the compilation.

The 'options(constant)' attribute can be specified for variables declared with the 'internal', 'static', and either the 'structure' or 'initial' attributes. If 'options(constant)' is specified for a structure, then all nonstructure members of the structure must have the 'initial' attribute.

It is an error to attempt to change the value of a variable declared with 'options(constant)' for the duration of a program.

CAPACITY OF STORAGE

The capacity of data storage of Multics PL/I is very large, but it does have limits. A program that exceeds the capacity of storage may be rejected by the compiler, may cause the occurrence of certain conditions that indicate storage overflow, may be interrupted by a Multics error message when storage overflows or may (in the worst case) proceed without detecting the error. Several different approaches can be taken to the capacity of storage, as follows:

- The programmer can assume that his use of storage is far below the capacity of storage and dismiss the further consideration of storage capacity. This view is convenient and correct for all but large-scale applications of Multics PL/I.
- At the other extreme, the programmer can seek to determine exactly the extent of his use of data storage and then compare this to an exact statement of storage capacity.
- As a compromise, the programmer can take into account only the use of data storage that consumes large quantities of storage and then compare that accounting to an approximate measure of storage capacity. This view is often sufficient for large-scale applications.

Storage Limits

The following figures are the approximate maximums for data storage, expressed in terms of 36-bit Multics words and given for different kinds of PL/I variables:

- 65,000 words (one quarter of a maximum segment) for automatic variables. This storage is called the stack.
- 500,000 words (approximately), for ordinary external static, internal static, controlled, and explicitly allocated based variables. This storage is called system storage.

Equivalenced based variables and defined variables use storage already allocated for other variables and so do not enter into a calculation of storage consumed.

Aside from the limits just given, there is one other limit on data storage. The storage for internal static variables is included as part of the object segment that is produced by the compilation of an external procedure. Therefore, the object text and the static internal variables cannot, taken together, use more than 262,000 words (one segment). When the Multics bind command is used to combine all of the object segments of a program into one object segment, all of the the static internal variables must fit into 16,384 words. Note that storage of the 'internal static' class can be simulated by allocating it in system storage and referencing it by an 'internal static' pointer.

Significant Uses of Storage

Because the limits of storage are so large, most uses of storage can be neglected. A program would have to declare thousands of scalar arithmetic variables, for example, before their use of storage would be significant. Only those items of storage that require hundreds or thousands of words need be considered, as follows:

- A variable with large extents is significant; included are an array variable with large bounds, a string variable with a large maximum length, and an area variable with a large size. Other variables are no more than four words long. No constant is longer than 64 words long (the maximum for a character-string constant).
- Any automatic, controlled, or based variable that is allocated hundreds or thousands of times becomes significant. This case can occur when an 'allocate' statement is part of a loop or when a procedure with automatic variables is executed recursively.

To determine whether a program remains within the limits of data storage capacity, determine the words used by the significant items of the program and compare the usage to the limits given for the different kinds of variables. The number of words used by a variable is determined by rules given in Section III, "Value Storage" under the heading "Storage Layout for Multics".

In practice, a large-scale program has a few very large arrays that are used as tables, and these arrays are the only items that need be considered in comparing the use of storage to the capacity of Multics PL/I.

CONDITIONS FOR STORAGE MANAGEMENT

As a result of the allocation operation, certain conditions can occur. The purpose of such a condition is to report that there is no storage available for use by the allocated variable. A program can include 'on' statements to respond to such a condition by performing a remedial action (such as freeing some storage). If the program does not have 'on' statements for this purpose, the PL/I processor outputs an error message and terminates program execution. Details are given later, in Section XIII, "Condition Handling".

'storage' Condition

When the 'storage' condition occurs, one of the following cases applies:

- The stack storage is nearly filled. The stack storage is used for activation regions. It can be exhausted either by allocation of a very large automatic variable or by a runaway recursive execution of a procedure.
- The system storage segment is nearly filled. System storage segment is used for ordinary external static variables, internal static variables, controlled variables and explicitly allocated based variables. The remedial action is to free some of these variables.

In most programs, no provision is made for remedial action for these conditions; that is, their occurrence is usually considered to be a programming error.

The 'storage' condition takes care of the classes of storage that can be allocated dynamically: automatic, controlled, and based. A condition is provided for these storage classes because they can cause storage overflow in one execution of a program but not in another. Storage can be allocated for one other class of storage: static. No condition is provided for that class because if overflow of static storage occurs for any execution of a program it will occur for all executions of the program, and therefore the program is invalid; furthermore, there is no way to free static storage and thus no remedial action for overflow.

'area' Condition

The 'area' condition occurs when an attempt is made to allocate storage in an area variable (that is, in an 'allocate' statement with an 'in' option) that cannot supply the storage. Two cases apply:

- An 'allocate' statement attempts to allocate a based variable for which there is no room in the area variable. A valid remedial action is to free some or all of the storage in the area variable (usually after copying the values into other storage or outputting them to a file). After this action, it is valid to resume program execution at the point of interruption.
- An assignment statement attempts to assign an area value to an area variable that is too small. The remedial action is to transfer to some other place in the program; it is not valid to return to the point of interruption.

The 'area' condition can play an important role in large scale programs for the manipulation of linked data structures.

SECTION VIII

EXPRESSIONS

An expression is used in a program wherever a value is required. Some expressions are very simple; for example, the variable reference 'x' and the constant literal '25' are both expressions. Some expressions combine several operators to calculate a value; for example, '2*alpha*(X-Y)' is an expression that contains three operators. Some expressions invoke procedures; for example, 'F(x)' yields a value that is obtained by invoking a procedure at entry point 'F' with the argument 'x'.

The rules for the interpretation of expressions are complicated. One source of complexity is the requirement that each expression have an associated storage type; there are many storage types, and the rules for their use vary from one kind of expression to another. Further complexity arises from the fact that expressions can have aggregate values. Yet another complication arises in the interpretation of references to 'based' and 'defined' variables.

A programmer who is learning PL/I can eliminate some of the complexity by ignoring features that he does not need. He may be able to avoid fixed-point arithmetic except for integer counters and subscripts; that simplifies the use of built-in functions and operators. He may be able to avoid aggregate expressions except, perhaps, for the assignment of one variable to another; that simplifies the rules for determining the storage type of expressions. He may be able to avoid 'based' and 'defined' variables; that eliminates the difficult rules for equivalencing variable names. By thus selecting a subset of the expressions, the programmer can skip over some of the more complicated rules.

In this section, the six kinds of expressions are listed and the features they have in common are described. Then each kind of expression is described separately and in detail.

GENERAL REMARKS ON EXPRESSIONS

An expression is one of the following six constructs:

- variable reference
- constant literal
- constant reference
- programmed function reference
- built-in function reference
- operator expression

Before these kinds of expressions can be discussed individually, some general features of expressions must be discussed. The following paragraphs consider the nesting and parenthesization of expressions, the determination of storage types, the use of aggregate expressions, and the rules for ordering and optimizing the evaluation of expressions. In the course of these general remarks, several examples of expressions are discussed in detail.

Nested Expressions

In some cases, an expression can contain other expressions. Specifically, an operator expression has expressions as its operands; a function reference has expressions as its arguments; a subscripted variable reference has expressions as its subscripts; and a locator-qualified variable reference has an expression as its locator-qualifier. Such a use of one expression within another is called nesting. There is no restriction on nesting, and an expression can contain an expression that contains another expression that contains another expression and so on, to any reasonable depth of nesting.

An example of nested expressions appears in the following program:

```
P:  proc;
    decl (x,u) float;
    decl (i,j) fixed;
    decl A(10,10) float;
    decl sin builtin;
    ...
    x = sin(u)*A(i+2,j);
    ...
    end;
```

The right-hand-side of the assignment statement is constructed of eight expressions, as follows:

- The entire expression is an operator expression, and it has 'sin(u)' and 'A(i+2,j)' as its operands.
- The expression 'sin(u)' is a built-in function, and it has 'u' as its argument.
- The expression 'A(i+2,j)' is a subscripted variable reference, and it has 'i+2' and 'j' as its subscripts.
- The expression 'i+2' is an operator expression, and it has 'i' and '2' as its operands.
- The expressions 'u' and 'i' and 'j' are simple variable references.
- The expression '2' is a constant literal.

Observe that the nesting in this example covers four levels: the product contains a subscripted variable reference that contains a sum that contains a simple variable reference.

Parenthesized Expressions

Any expression can be enclosed in parentheses. There are two situations in which the use of parentheses is appropriate, as follows:

- A parenthesized expression is used to modify the order in which operators are evaluated. For example, consider the expression:

2*(a+b)

The operator priority rules of PL/I, given later in this section, specify that the '*' operation is performed before the '+' operation. However, the parentheses in this expression cause the '+' operation to be performed first.

- A parenthesized expression is sometimes used as an argument in a function reference or a 'call' statement; specifically, it is used when it is necessary to force an argument that would otherwise be interpreted as by-reference to be interpreted as by-value. This is a specialized use of a parenthesized expression; it is discussed in Section XII, "Procedure Invocation."

The storage type and value of a parenthesized expression are the same as those of the enclosed expression.

Storage Types of Expressions

The interpretation of an expression yields a value and storage type. The value of an expression is determined each time the expression is evaluated during program execution. In contrast, the storage type is determined once, by the compiler, before the program is executed.

Storage types were defined earlier, in Section III, "Value Storage." The storage type of an expression determines the way the value of an expression is represented. Because the storage type is determined in advance, the compiler can provide exactly the required storage for the value and can generate instructions that are appropriate for the value.

As a basis for the discussion of the storage types of expressions, consider the following program:

```
P:  proc;
    dcl i fixed;
    dcl (x,theta) float;
    dcl B(20) float;
    dcl cos builtin;
    ...
    x = B(i-3)*cos(theta);
    ...
end;
```

The right-hand-side of the assignment statement is made up of seven expressions. The storage type of each expression is obtained as follows:

- The storage type of the variable reference 'i' is 'real fixed bin(17,0)'. It is obtained from the first 'declare' statement according to the rules for a simple variable reference given later in this section.
- The storage type of the constant literal '3' is 'real fixed dec(1,0)'. It is obtained from inspection of the constant literal itself according to rules given later in this section. Those rules specify that the constant literal is 'real' because it does not have an 'i' at the end, 'fixed' because it does not have an exponent, 'decimal' because it does not have a 'b' at the end, and 'precision(1,0)' because it has one digit and that digit is not a fractional digit.
- The storage type of the expression 'i-3' is 'real fixed bin(18,0)'. It is obtained from the rules for the '-' operator, which are given later, in Section IX, "Operations." According to those rules, the constant literal is first converted to a 'real fixed bin(4,0)' value; this can be done during compilation. The number-of-digits of the result is 18 in order to allow for the possibility of a carry from the subtraction operation.
- The storage type of the subscripted variable reference 'B(i-3)' is 'real float bin(27)'. It is obtained from the third 'declare' statement according to the rules given later in this section. The subscript in the reference cancels out the extent in the declaration, so that the storage type of the reference is scalar.
- The storage type of the variable reference 'theta' is 'real float bin(27)'. It is obtained from the second 'declare' statement according to the rules for a simple variable reference.
- The storage type of the built-in function reference 'cos(theta)' is 'real float bin(27)'. It is obtained from the rules for the 'cos' built-in function, which are given later, in Section IX, "Operations." In this simple case, the storage type of the result is the same as the storage type of the argument.
- The storage type of the entire expression is 'real float bin(27)'. It is obtained from the rules for the '*' operation, which are given later, in Section IX, on "Operations." Observe that the storage type of the result is, in this case, the same as the storage type of the operands.

This example is typical in several respects. It shows that even a simple use of fixed-point arithmetic has some tricky points. It shows the simplicity of floating-point calculations, which often carry the single storage type 'real float bin(27)' straight through the calculation. And it shows how an aggregate is referenced to obtain a scalar value.

Aggregate Expressions

The main part of this section is devoted to defining the six kinds of PL/I expressions. In the definition of each kind of expression, aggregate values are mentioned. That aspect of the definition of expressions is summarized here:

- A variable reference or a programmed function reference can have any storage type and therefore can have any aggregate type. This generality permits aggregate values to be handled as single entities when they are subjected to input/output, are copied from one variable to another, or are operated upon by a procedure.
- A constant literal or a constant reference cannot (except for 'label' constant references) have an aggregate type because PL/I does not include a way of writing an aggregate constant. This omission reflects a decision of the designers of PL/I rather than a fundamental limitation of programming languages. It is quite easy to program around this deficiency.
- Most built-in function references and operator expressions can have aggregate types; in such a case, the aggregate type of the result is derived from the aggregate types of the arguments or operands. Operations on aggregates are performed on their respective components; that is, an operation is applied to the ith component of each argument to produce the ith component of the result.

The aggregate variable references and programmed function references can be especially useful in writing a clear and efficient program. In contrast, the aggregate built-in function references and operator expressions are less often useful.

Ordering and Optimizing the Evaluation of Expressions

The definition of PL/I deliberately leaves undefined certain aspects of expression evaluation. In fact, the only general rule for expression evaluation is:

- When the value of an expression is required, it is evaluated at some time after the last computation that could change the value of the expression and at some time before the computation that requires its value.

The vagueness of this rule permits the compiler to determine the details of the evaluation of an expression and thus produce optimized code for the program. For example:

- When the same expression appears in several places, the compiler can evaluate it just once if the compiler can determine that the value of the expression does not change between the given appearances.
- The subscripts of a variable reference can be evaluated in any order. The same freedom applies to the arguments of a function reference or the operands of an operator expression.
- An argument or operand can be ignored if it does not affect the result of an operation. For example, the second operand of an "and" operation can be ignored when the value of the first operand is "false".

Thus there can be more than one way to evaluate a given expression. However, the variations that are permitted are chosen so that, in most cases, they have no effect on the results.

Some consideration must be given to the cases in which the undefined aspects of expression evaluation do affect the results of program execution. These cases arise because of side effects. A side effect is a change in the value of a variable or in the environment of the program that is caused by the evaluation of an expression. There are two kinds of side effects:

- The evaluation of a programmed function reference can invoke a procedure that produces a side effect.
- The evaluation of an expression can cause a condition to occur that signals an 'on' unit that produces a side effect.

It is easy to recognize the possibility of a side effect produced by a programmed function reference. It is not so easy to fully appreciate the possibilities of side effects from the occurrence of conditions. Such conditions as 'size', 'fixedoverflow', 'underflow', and 'overflow' can occur almost anywhere in the evaluation of an expression. Thus the side effects caused by an 'on' unit are especially significant.

As an example of a program whose results are partially undefined, consider the following:

```
P:  proc;
    dcl (x,y) float;
    dcl (sysin,sysprint) file;
    get list(x,y);
    put list(F(x)+F(y));
F:  proc(a) returns(float);
    dcl a float;
    put list(a);
    return(a**2);
    end;
end;
```

The first 'put' statement in this program must evaluate the expression:

$F(x)+F(y)$

Each operand in this expression is a programmed function reference that has a side effect. The side effect is the listing of the value of the argument of the programmed function reference; that action changes the environment of the program. Because the order in which the operands of '+' are evaluated is undefined, the order in which the argument values are listed is undefined. Therefore, the result of executing the program is partially undefined. Nevertheless, the program is valid and it is also correct unless the programmer cares about the order in which 'x' and 'y' are listed.

The results of executing a program are not necessarily undefined just because some part of the execution of the program is undefined. For example, suppose the statement:

```
put list(a);
```

is removed from the program given in the previous paragraph. After this change, the evaluations of the programmed function references have no side effects. The execution of the program is still undefined because the order in which the operands of '+' are evaluated is still undefined. However, the result of executing the program is fully defined: the program lists a single value that is the sum of the squares of 'x' and 'y'.

VARIABLE REFERENCES

The interpretation of a variable reference begins with the determination of two items of information: the location of a variable in storage and the storage type of the variable. For some variable references, this information is easy to obtain; for others, it requires a careful interpretation of the reference. Once this information has been obtained, the remainder of the interpretation of the variable reference depends on whether the variable reference is used as an expression or as the target of an assignment. If the variable reference is used as an expression, then the value of the designated variable is retrieved and becomes the value of the reference. If the variable reference is used as the target of an assignment, then the assigned value is converted to the designated type and is placed in the designated variable.

Variable Reference Types

There are four kinds of variable references, as follows:

- simple
- subscripted
- structure-qualified
- locator-qualified

This list is given in order of increasing complexity and decreasing frequency of use. Thus, for example, a locator-qualified variable reference has a complicated interpretation but is used only in special programming applications.

Each of the four kinds of variable reference is described in detail in this section. Each description depends on the preceding descriptions; for example, the description of subscripted references makes use of rules that are given in the description of simple references.

MAJOR NAME IN A VARIABLE REFERENCE

A variable reference begins with a major name unless it is shortened or locator-qualified. The major name designates a variable in storage that is not part of a larger variable, and the remainder of the reference indicates the portion of the variable that must be retrieved. Consider, first, a simple variable reference:

Speed3

In this case, the major name is the entire variable reference, so the value of the reference is the value of the entire variable. Consider, next, a subscripted variable reference:

A(3)

In this case, the major name 'A' designates an array variable and '(3)' designates an element of that array variable. Consider, finally, a structure-qualified variable reference:

alpha.Q(i,j-3)

In this case, the major name 'alpha' designates a structure variable and '.Q(i,j-3)' designates an element of a two-dimensional array that is a member of the structure variable.

When a variable reference is shortened, the major name may be missing. For example, in certain contexts the structure-qualified variable reference given in the preceding paragraph can be shortened to:

Q(i,j-3)

In this case, 'Q' is not the major name. Instead, the reference must be expanded to its original, complete form before it can be interpreted; then 'alpha' is the major name, as before.

The interpretation of a locator-qualified variable reference is quite different from the other three kinds of variable reference. In a locator-qualified reference, the place of the major name is taken by a locator-qualifier. The locator-qualifier, like the major name, designates a variable in storage; however, it can be any reference that yields a locative value. Thus the designation is computed at the time the reference is interpreted instead of being given, as a name, once and for all when the program is written. This general facility is useful for list-processing applications of PL/I.

Simple Variable References

A simple variable reference designates a major variable. The variable can be either a scalar or an aggregate.

FORM OF SIMPLE VARIABLE REFERENCES

A simple variable reference has the following form:

id

where id must be an identifier that is declared as the name of a major variable and that is not declared 'based'.

As an example of a simple variable reference, consider:

alpha

This name must have a declaration of the form:

```
dcl alpha ... ;
```

or the form:

```
dcl 01 alpha ... ,  
    02 ...  
    ... ;
```

where the '...' symbols indicate portions of the declarations that are not significant for this discussion.

INTERPRETATION OF SIMPLE VARIABLE REFERENCES

A simple variable reference that is used for retrieval is interpreted as follows:

1. Name Resolution. Resolve the variable name. (The resolution of names is described earlier, in Section VI, "Declarations.") The result is the declaration of the variable name.
2. Storage Type Determination. Obtain the storage type of the variable name from its declaration. The result is the storage type for the given reference.
3. Variable Location. Locate the variable designated by the variable name.
4. Value Retrieval. Retrieve the value of the designated variable.

Most of the interpretation is performed by the compiler; only the last step, value retrieval, is performed during program execution.

The location of the variable, mentioned in Step 3 of the interpretation, depends not only on the name of the variable but also on its storage class and scope attributes, as follows:

- If the name is 'static external', then the variable is in the external region.
- If the name is 'static internal', then the variable is in the permanent internal region associated with the block in which the name is declared.
- If the name is 'controlled external', then the variable is in the external region. If more than one generation has been allocated for the given name, then the most recently allocated generation is used.
- If the name is 'controlled internal', then the variable is in the permanent internal region associated with the block in which the name is declared. If more than one generation has been allocated for the given name, then the most recently allocated generation is used.
- If the name is 'automatic internal', then the variable is in the activation internal region that corresponds to the block in which the name is declared. If the program is recursive, then there can be more than one activation internal region for a given block; the selection of one of these regions is made according to rules given in Section XII, "Procedure Invocation."
- If the name is declared 'parameter internal', then the variable is reached by first locating a 'pointer' temporary that is designated by the name and that is in the activation internal region that corresponds to the block in which the name is declared. Then the pointer is followed to the desired variable. Details are given in Section XII, "Procedure Invocation."
- If the name is declared 'defined internal', then the variable is located by rules that are given in Section VII, "Storage Management."

Once the appropriate storage region has been located, the variable is uniquely designated by the given name.

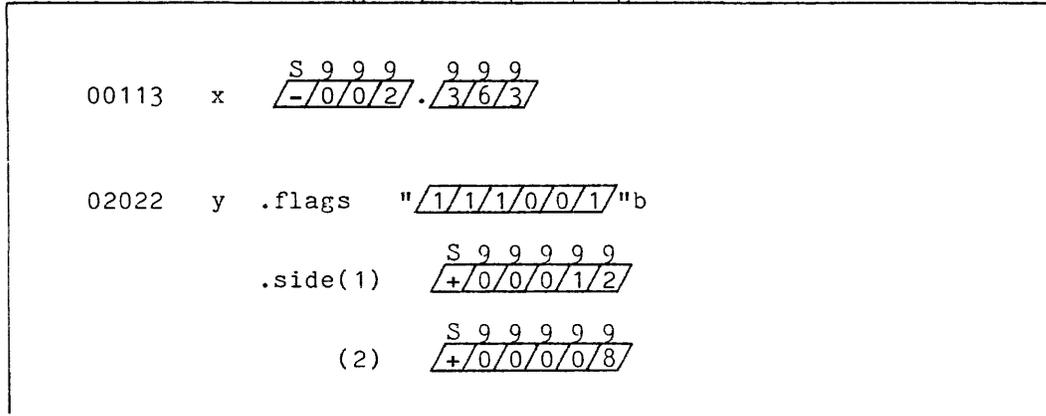
EXAMPLES OF SIMPLE VARIABLE REFERENCES

Examples of simple variable references are discussed here; they occur in the following program:

```
P:  proc;
    dcl x dec(6,3);
    dcl 01 y,
        02 flags bit(6),
        02 side(2) dec(5);
    dcl (sysin,sysprint) file;
    ... (assignments to variables occur here)
    put list(x,y,flags,side);
    ...
end;
```

When execution of the program begins, the variables 'x' and 'y' are allocated; then, when execution of the program is underway, values are assigned to the variables. Suppose that before the example output statement is executed, storage includes the region:

activation internal region, example program



Four variable references appear in the output statement in the example program:

- o 'x' is a true simple reference (that is, it is not a shortened form of some other reference) and it yields:
 - storage type: real fixed dec(6,3)
 - value: -2.363
- o 'y' is also a true simple reference, and it yields:
 - storage type: 01, 02 bit(6) nonvarying,
02 dim(2) real fixed dec(5,0)
 - value: "111001"b, 12, 8
- o 'flags' is a shortened form of the structure-qualified reference 'y.flags' and is interpreted accordingly (as described under "Structure-Qualified Variable References" later in this section).
- o 'side' is a shortened form of 'y.side(*)', which also is a structure-qualified reference.

Subscripted Variable References

A subscripted variable reference designates a portion of a major array variable. The designated portion can be either a single element of the array, in which case it is a scalar, or a cross section of the array, in which case it is, itself, an array. The designated portion of the major variable is specified by one or more subscripts.

FORM OF SUBSCRIPTED VARIABLE REFERENCES

A subscripted variable reference has the following form:

```
mn( sublist )
```

where mn is the major name and sublist is the subscript list. The major name must be an identifier that is declared as the name of a major variable and that is not declared 'based'. The subscript list is a sequence of subscripts separated by commas, and each subscript is either an expression or an '*'. A subscript expression must yield a value that can be converted to an integer. A subscripted variable reference that has one or more '*' subscripts designates a cross-section of an array.

As an example of a subscripted variable reference, consider:

```
PHI(i+ceil(.362*sqrt(x-1)),*,j-2)
```

In this example the major name is 'PHI' and there are three subscripts. The first subscript is chosen to show that there is no special restriction on a subscript expression. The major name must have a declaration of the form:

```
decl PHI( dim , dim , dim ) ... ;
```

where each dim represents a dimension and '...' represents a sequence of attributes.

INTERPRETATION OF SUBSCRIPTED VARIABLE REFERENCES

A subscripted variable reference that is used for retrieval of a value is interpreted as follows:

1. Name Resolution. Resolve the major name in the given reference. The result is the declaration of the major name.
2. Storage Type Determination. Make a copy of the storage type of the major name and delete from the 'dimension' attribute of the name each dimension that corresponds to a subscript that is an expression. Do not delete a dimension that corresponds to a '*' subscript. If all dimensions are deleted, then delete the 'dimension' attribute. The result is the storage type of the reference.
3. Subscript Evaluation. Evaluate each subscript expression in the reference and, if necessary, convert its value to an integer. If a subscript value is outside the range of subscripts for which the array variable is allocated, the 'subscriptrange' condition occurs. The result of subscript evaluation is the fully-bound reference.

4. Variable Location. Locate the variable designated by the major name; do this just as for a simple variable reference. The designated variable is the sequence of storage units that are selected by the fully bound reference. A storage unit is selected if it matches the beginning of the designator or the entire designator of the storage unit. The match must be exact except that a '*' in the reference matches any integer subscript in a storage unit designator.
5. Value Retrieval. Retrieve the value of the designated variable.

Most of the interpretation is performed by the compiler; only subscript evaluation and value retrieval are performed during program execution.

The interpretation just given for a subscripted variable reference is complete, but it requires the following remarks to clear up difficult points:

- The determination of the storage type by Step 2 and the retrieval of the value by Step 4 are consistent with one another. That is, the interpretation of the reference guarantees that the value retrieved is always appropriate for the storage type of the reference.
- The 'subscriptrange' condition mentioned in Step 3 is part of the mechanism provided by the language to detect errors or exceptions that occur during program execution. The programmer can supply statements to handle such a condition; more often, he allows the interpreter to report it as an error and abort the program. There is a cost associated with checking the value of a subscript, and the language allows the programmer to conveniently enable this check during program debugging and then disable it when the program enters production. Details are given later, in Section XIII, "Condition Handling."
- In general, a subscripted variable reference selects a subset of the elements of the array variable that is designated by the major name. When a subscript is an expression (and is evaluated to produce an integer), it participates in making the subset smaller. On the other hand, when a subscript is a '*', it makes no contribution to the selection and allows any subscript value to match its subscript position.

Two special cases of the subscripted variable reference are of particular interest:

- The most common use of a subscripted variable reference is that in which the major name is declared to be an array of scalars and the subscript list contains no '*'. In this case, the storage type of the reference is scalar and the fully-bound reference designates a single storage unit.
- When every subscript is a '*', the reference designates the entire variable designated by the major name. Specifically, it follows from Step 2 that no dimension is deleted from the storage type and it follows from Step 4 that the fully-bound reference matches the designator of every storage unit of the array variable.

EXAMPLES OF SUBSCRIPTED VARIABLE REFERENCES

Examples of subscripted variable references are discussed here; they occur in the following program:

```
P:  proc;
    decl A(3,2) dec(4);
    decl 01 part(0:1),
          02 name char(6),
          02 code dec(5);
    decl sysprint file;
    decl (i,j,m) fixed;
    decl x float;
    ... (assignments to variables occur here)
    put list(A(i+2,j-x**2),A(i+2,*),A(*,j-x**2),A(*,*));
    put list(part(m),part(*));
    ...
end;
```

Suppose that before the example output statements are executed, storage includes the region:

activation internal region, example program

00242	A	(1,1)	<u>S 9 9 9 9</u> <u>+/0/2/0/1/</u>
00244		(1,2)	<u>S 9 9 9 9</u> <u>+/0/2/0/2/</u>
00246		(2,1)	<u>S 9 9 9 9</u> <u>+/0/2/0/3/</u>
00250		(2,2)	<u>S 9 9 9 9</u> <u>+/0/2/0/4/</u>
00252		(3,1)	<u>S 9 9 9 9</u> <u>+/0/2/0/5/</u>
00254		(3,2)	<u>S 9 9 9 9</u> <u>+/0/2/0/6/</u>
03544	part	(0).name	<u>X X X X X X</u> <u>"t/x/8/Q/2/4"</u>
03546		.code	<u>S 9 9 9 9 9</u> <u>+/0/0/5/9/3/</u>
03550		(1).name	<u>S 9 9 9 9 9</u> <u>"w/6/r/f/E/D"</u>
03552		.code	<u>S 9 9 9 9 9</u> <u>+/8/0/0/0/7/</u>

Subscripted references to the same variable differ in an important way when one reference has an expression for a subscript and the other has a '*' for the same subscript. According to this view, four different references can be written for the variable 'A', as follows:

- 'A(i+2,j-x**2)' has expressions as subscripts. Suppose that the reference is interpreted when:

i = 0, j = 5, and x = 2

Then the fully-bound reference is 'A(2,1)'. The result of the interpretation of the reference is:

storage type: real fixed dec(4,0)
value: 203

- 'A(i+2,*)' is a one-dimensional cross-section of the two dimensional array variable. Suppose the value of 'i' is as before; then the fully-bound reference is 'A(2,*)'. The result of the interpretation of the reference is:

storage type: dim(2) real fixed dec(4,0)
value: 203,204

- 'A(*,j-x**2)' is also a one-dimensional cross-section of the two dimensional array variable. Suppose the values of 'j' and 'x' are as before; then the fully-bound reference is 'A(*,1)'. The result of the interpretation of the reference is:

storage type: dim(3) real fixed dec(4,0)
value: 201, 203, 205

- 'A(*,*)' is a two-dimensional cross-section of the two dimensional array variable; that is, it designates the entire array variable. The result of the interpretation of the reference is:

storage type: dim(3,2) real fixed dec(4,0)
value: 201, 202, 203, 204, 205, 206

The variable 'part' is an array of structures. There are two different references for the variable, as follows:

- 'part(m)' has an expression as its subscript. Suppose the reference is evaluated when:

m = 0

Then the fully-bound reference is 'part(0)'. The result of the interpretation of the reference is:

storage type: 01, 02 char(6) nonvarying,
02 real fixed dec(5,0)

value: "tx8Q24", 593

- 'part(*)' designates the entire variable. The result of interpretation of the reference is:

storage type: 01 dim(1), 02 char(6) nonvarying,
02 real fixed dec(5,0)

value: "tx8Q24", 593, "w6rfED", 80007

Structure-Qualified Variable References

A structure-qualified variable reference designates a portion of a variable that is a structure or an array of structures. The designated portion of the variable can either be a scalar or another, smaller, aggregate; it is selected by level names and, in some cases, subscripts.

FORM OF STRUCTURE-QUALIFIED VARIABLE REFERENCES

A structure-qualified variable reference has one of the following forms:

```
lref1 . lref2  
lref1 . lref2 . lref3  
... (and so on)
```

where lref1, lref2, lref3, and so on, are level references. Each level reference is a level name optionally followed by a parenthesized subscript list. The subscript list is a sequence of subscripts separated by commas, and each subscript is either an expression or an '*'. A subscript expression must yield a value that can be converted to an integer.

The rightmost level reference is the member reference and the other level references are containing references. The leftmost containing reference must begin with a major name that designates an aggregate variable. The structure-qualified reference as a whole must be consistent with the declaration of the major name; that is, the first level reference must designate a second-level component of the major variable, the second-level reference must designate a third level component, and so on.

An example of a structure-qualified reference is:

```
base.first
```

In this example, there are two level references, each in the form of a simple reference. The name 'base' is a containing reference and is the major name for the structure-qualified reference as a whole. The name 'first' is the member reference. The identifiers must be declared by a statement of the form:

```
dcl 01 base ... ,  
    ...  
    02 first ...,  
    ... ;
```

where the '...' symbols indicate portions of the statement that are not of interest here.

A second and more complicated example of a structure-qualified reference is:

```
x(j,*).y3_test(2*m-3/i).par4
```

In this example, there are three level references: the first two, the containing references, are in the form of subscripted references and the last one, the member reference, is in the form of a simple reference. The name 'x' is the major name for the reference and it must be declared as a level-one, two-dimensional array of structures. The name 'y3_test' must be declared as a one-dimensional array of structures that is a member of 'x'. The identifier 'par4' must be declared as a member of 'y3_test'. In other words, the following 'declare' statement must apply:

```
dcl 01 x( dim , dim ) ... ,
      ... ,
      02 y3_test( dim ) ... ,
      ... ,
      03 par4 ... ,
      ... ,
      ... ;
```

Observe that although the reference places constraints on the aggregate type of the containing references, it places no constraint on the member reference 'par4'; so 'par4' could be a scalar, array, or structure.

INTERPRETATION OF STRUCTURE-QUALIFIED VARIABLE REFERENCES

A structure-qualified reference that is used for retrieval of a value is interpreted as follows:

1. Name Resolution. Resolve the major name in the given reference. The result is the declaration of the major name.
2. Storage Type Determination. Perform the following steps:
 - a. Make a copy of the normalized storage type for the major name of the given reference.
 - b. Edit the dimensions in the storage type as follows:
 - (1) Delete each dimension that is associated with a subscript expression in the given reference.
 - (2) Keep each dimension that is associated with a '*' subscript in a containing reference of the given reference, but move it so that it occurs in the storage-type component that corresponds to the member reference.
 - (3) Keep all other dimensions.
 - c. Edit the remainder of the storage type as follows:
 - (1) Keep the component of the storage type that corresponds to the member reference.
 - (2) If the member reference designates a structure, keep each component of the storage type that corresponds to a component of that structure.
 - (3) Delete the remaining components of the storage type.

The result is the storage type of the reference. It is understood that, as the storage type is derived by the steps above, the necessary refinements are supplied to keep the storage type in a valid, normalized form. For example, when the last dimension in a 'dimension' attribute is deleted, the 'dimension' attribute itself is deleted.

3. Subscript Evaluation. Subscripts are evaluated just as already described for a subscripted variable reference. The result is the fully-bound reference.
4. Variable Location. Locate the variable designated by the given major name; do this just as for a simple variable reference. The designated variable is the sequence of storage units that are selected by the fully bound reference. A storage unit is selected by the fully bound reference. A storage unit is selected if it matches the beginning of the designator or the entire designator of the storage unit. The match must be exact except that a '*' in the reference matches any integer subscript in a storage unit designator.
5. Value Retrieval. Retrieve the value of the designated variable.

Most of the interpretation is performed by the compiler; only subscript evaluation and value retrieval are performed during program execution.

This interpretation, especially the determination of the storage type in Step 2, is complicated. Rather than discuss it in the abstract, a detailed discussion of an example reference is given in what follows.

EXAMPLES OF STRUCTURE-QUALIFIED VARIABLE REFERENCES

Examples of structure-qualified references are discussed here; they occur in the following program:

```
P:  proc;
      dcl 01 Q(2) static external,
          02 R1,
          03 S1(4) float,
          03 S2 char(4),
          02 R2(0:1,3) dec(10);
      dcl sysprint file;
      ... (assignments to variables occur here)
      put list(Q(*).R1.S1(i+3));
      ...
      end;
```

A diagram of the storage for 'Q' would be of inconvenient size. Instead, a complete list of the designators for the storage units that make up the variable is given:

```
Q(1).R1.S1(1), Q(1).R1.S1(2), Q(1).R1.S1(3), Q(1).R1.S1(4),
Q(1).R1.S2,
Q(1).R2(0,1), Q(1).R2(0,2), Q(1).R2(0,3),
Q(1).R2(1,1), Q(1).R2(1,2), Q(1).R2(1,3),
Q(2).R1.S1(1), Q(2).R1.S1(2), Q(2).R1.S1(3), Q(2).R1.S1(4),
Q(2).R1.S2,
Q(2).R2(0,1), Q(2).R2(0,2), Q(2).R2(0,3),
Q(2).R2(1,1), Q(2).R2(1,2), Q(2).R2(1,3)
```

The interpretation of a structure-qualified reference is now given in detail. Suppose the reference:

```
Q(*).R1.S1(i+3)
```

is interpreted when $i = -1$. The steps in the interpretation are:

1. (Name Resolution.) The 'declare' statement in the example program is associated with 'Q'.
2. (Storage Type Determination.) The following steps are performed:
 - a. A copy of the normalized storage type is made, giving:

```
01 dim(2),
02,
03 dim(4) float,
03 char(4),
02 dim(2,3) dec(10)
```

Observe that the normalized form of the dimension '0:1' (declared for 'R2') is '2'.

- b. The dimension is edited. The dimension associated with 'i+3' is omitted, giving:

```
01 dim(2),
02,
03 float,
03 char(4),
02 dim(2,3) dec(10)
```

Then the dimension associated with the '*' subscript in the first containing reference is moved to the storage type component associated with the member reference, giving:

```
01,
02,
03 dim(2) float,
03 char(4),
02 dim(2,3) dec(10)
```

If this dimension were not moved, it would be deleted by Step 2c, and that would be inconsistent with the interpretation of a '*' subscript.

- c. The remainder of the storage type is edited. The component of the storage type that corresponds to the member reference, 'S1(i+3)' is kept. Since 'S1' is not a structure, nothing else is kept. The result is:

```
03 dim(2) float
```

A level number on a storage type that is not a structure is not permitted, so the final result is:

```
dim(2) float
```

3. (Subscript Evaluation.) The subscript expression in the given reference is evaluated. The fully-bound reference is:

Q(*).R1.S1(2)

4. (Value Retrieval.) An inspection of the 22 designators for the scalar components of 'Q' shows that the fully-bound reference matches two, namely:

Q(1).R1.S1(2)
Q(2).R1.S1(2)

The sequence of two scalar values associated with these designators is retrieved and is the value of the given reference. Observe that the storage type of this result is identical to the storage type obtained in Step 2.

If the difference between two subscript expressions is ignored and if shortened references are excluded, then there are 18 distinct references to the variable designated by 'Q', as follows:

Q(*)	Q(i)
Q(*)R1	Q(i).R1
Q(*)R1.S1(*)	Q(i).R1.S1(*)
Q(*)R1.S1(j)	Q(i).R1.S1(j)
Q(*)R1.S2	Q(i).R1.S2
Q(*)R2(*,*)	Q(i).R2(*,*)
Q(*)R2(*,k)	Q(i).R2(*,k)
Q(*)R2(j,*)	Q(i).R2(j,*)
Q(*)R2(j,k)	Q(i).R2(j,k)

One of these forms of reference, 'Q(*)R1.S1(j)', has just been considered in detail. Several other forms are now considered:

- Q(i).R1.S1(j) (assume i = 1 and j = 3)

storage type: float

designator: Q(1).R1.S1(3)

(Out of the 18 forms of reference for 'Q', only three have scalar values. This is one of them.)

- Q(i).R2(j,k) (assume i = 1, j = 0, k = 2)

storage type: dec(10)

designator: Q(1).R2(0,2)

(This is the second form that has a scalar value. The third is 'Q(i).R1.S2'.)

- Q(*)

storage type: 01 dim(2),
02,
03 dim(4) float,
03 char(4),
02 dim(2,3) dec(10)

designators: (The full sequence of 22 designators)

(This reference designates the entire variable associated with 'Q'.)
- Q(i).R1 (assume i = 2)

storage type: 01,
02 S1(4) float,
02 S2 char(4)

designators: Q(2).R1.S1(1), Q(2).R1.S1(2),
Q(2).R1.S1(3), Q(2).R1.S1(4)
Q(2).R1.S2

(This storage type is essentially the declaration of 'R1'; only the level numbers have been changed.)
- Q(*) .R1.S1(*)

storage type: dim(2,4) float

designators: Q(1).R1.S1(1), Q(1).R1.S1(2),
Q(1).R1.S1(3), Q(1).R1.S1(4),
Q(2).R1.S1(1), Q(2).R1.S1(2),
Q(2).R1.S1(3), Q(2).R1.S1(4)

(Here, two separate dimensions combine to make a two-dimensional array.)
- Q(i).R2(*,*) (assume i = 2)

storage type: dim(2,3) dec(10)

designators: Q(2).R2(0,1), Q(2).R2(0,2), Q(2).R2(0,3),
Q(2).R2(1,1), Q(2).R2(1,2), Q(2).R2(1,3)

(Observe that the dimension of 'R2', which is '0:1', is normalized to '2'; but this does not affect the subscripts used in the designators.)
- Q(*) .R2(j,*) (assume j = 0)

storage type: dim(2,3) dec(10)

designators: Q(1).R2(0,1), Q(1).R2(0,2), Q(1).R2(0,3),
Q(2).R2(0,1), Q(2).R2(0,2), Q(2).R2(0,3)

(Here, the storage type is exactly the same as for the previous example; but the sequence of designators, and therefore the value, is different.)

Locator-Qualified Variable References

A locator-qualified variable reference makes use of a 'pointer' or 'offset' value to locate a variable. Once the variable is located, it can be accessed by any of the means thus far described in this discussion of variable references.

FORM OF A LOCATOR-QUALIFIED VARIABLE REFERENCE

A locator-qualified variable reference has the following form:

lq -> br

where lq is the locator qualifier and br is the based reference. The locator qualifier must be a reference that yields a 'pointer' or 'offset' value. The based reference must have the form of a simple variable reference, a subscripted variable reference, or a structure-qualified variable reference; however, the major name of the based reference must be declared 'based'.

A simple example of a locator-qualified reference is:

p->x

In this example, 'p' must be declared 'pointer' or 'offset', and 'x' must be declared 'based'. In an English reading of a program, the reference can be expressed as "p arrow x" or, more descriptively, as "the value obtained by interpreting the value of 'p' as a pointer to a variable that has the storage type given by 'x'".

Other examples of locator-qualified references are:

F(x+2,3*m)->Y

q.alpha(j,k).r2->top(i+3).next

f->g->h

In the first example, the locator qualifier is a function reference or a subscripted reference (depending on the declaration of 'F') that must be declared 'pointer' or 'offset'. In the second example, both the locator qualifier and the based reference are complicated structure-qualified references. In the third example, the locator qualifier for the entire reference is 'f->g', and the locator qualifier for 'f->g' is 'f'. For a given reference, it is always the rightmost arrow that separates the locator qualifier from the based reference.

ASSOCIATED STORAGE TYPES FOR LOCATOR VALUES

Every locator value has an associated storage type. This storage type is not the storage type of the locator value itself; that storage type is either 'pointer' or 'offset'. Instead, the associated storage type is the storage type of the variable that is designated by the locator value.

The associated storage type is created when the locator value is created. Two cases apply:

- A locator value is created as the result of the application of the 'addr' built-in function to a given variable reference. In this case, the associated storage type is the storage type of the given variable reference.
- A locator value is created when an 'allocate' statement is executed on a given variable name; the statement causes a locator value to be assigned (through a 'set' option) to a locator variable. In this case, the associated storage type is the storage type of the given variable name.

The associated storage type accompanies a locator value wherever it goes: as the value is assigned from one variable to another or as the value is returned by a programmed function reference.

As an example of the handling of associated data types, consider the following program:

```
P:  proc;
    dcl (p1,p2,p3,p4) pointer;
    dcl a(10) float;
    dcl 01 Q based,
        02 R1 float,
        02 R2 dec(5);
    ...
    allocate Q set(p1);
    ...
    p2 = addr(a);
    ...
    p3 = addr(a(3));
    ...
    p4 = p2;
    ...
end;
```

After the 'allocate' statement, the 'pointer' variable designated by 'p1' contains a pointer value; and that value has the associated storage type '01, 02 float, 02 dec(5)'. After the first assignment statement, 'p2' has a pointer value whose associated storage type is 'dim(10) float'. After the second assignment statement, 'p3' has a pointer value whose associated storage type is 'float'. After the last assignment statement, 'p4' has a pointer value whose associated storage type is 'dim(10) float'.

INTERPRETATION OF LOCATOR-QUALIFIED VARIABLE REFERENCES

A locator-qualified variable reference that is used for retrieval of a value is interpreted as follows:

1. Name Resolution. Resolve the major name in the based reference of the given reference. The result is the declaration of the major name.
2. Locator-Qualifier Evaluation. Evaluate the locator qualifier for the given reference. The result is a pointer value or an offset value; if it is an offset value, convert it to a pointer value. The result is the base pointer value for the given reference.
3. Base-Variable Location. Use the base pointer value to locate a position in storage. At that position, a variable begins whose storage type is the same as the associated storage type of the base pointer value. This variable is the base variable for the given reference.
4. Based-Variable Overlay. Overlay a based variable on the base variable. That is, define the set of designators that would have been produced if the base variable had been allocated in accordance with the declaration of the major name of the based reference.

5. Based Reference Interpretation. Interpret the based reference that occurs in the given reference. That is, determine its storage type, evaluate its subscripts, and evaluate it. The results are the storage type and value for the entire locator-qualified reference.
6. Based-Variable Withdrawal. Withdraw the based variable from the base variable; that is, discard the designators that were defined in Step 4.

Most of the interpretation is performed by the compiler; only locator-qualifier evaluation (Step 2) and a portion of based reference interpretation (Step 5) are performed during program execution.

According to Step 3 of the interpretation, the associated storage type of the base pointer value must agree with the storage type of the major name in the based reference of the given reference. The Multics implementation of PL/I (as well as other implementations) does not check for a violation of this rule. Because of a deficiency in the design of PL/I, the check cannot be made at compile time. Because the cost would be unreasonably high, the check is not made at execution time. Therefore, the programmer must detect his own errors in this respect.

EXAMPLES OF LOCATOR-QUALIFIED VARIABLE REFERENCES

Examples of locator-qualified variable references are discussed here; they occur in the following program:

```
P:  proc;
    dcl (n,k) fixed;
    dcl 01 H(100) static,
        02 flags bit(3),
        02 item(2),
        03 text char(6),
        03 count dec(5),
        02 style dec(4);
    dcl 01 i(n) based,
        02 t char(6),
        02 c dec(5);
    dcl P1 pointer;
    dcl sysprint file;
    ... (assignments to variables occur here)
    put list(P1->i(2*k-4));
    ...
end;
```

Just before the examples of retrieval are interpreted, suppose that storage includes the regions:

permanent internal region, example program

```

17762  H (1).flags  "1 1 1/1/1/0"b
(... and so on for the first 7 elements of 'H')

20144  H (8).flags  "1 1 1/1/1/0"b

20145  ---- .item(1).text "x x x x x x/Q/n/a/i/l/s"

20147  ----- .count  S 9 9 9 9 9/+00144

20151  ----- (2).text "x x x x x x/a/C/C/i/d/e"

20153  ----- .count  S 9 9 9 9 9/+00090

20155  ---- .style  S 9 9 9 9/-0013

(... and so on for the remaining 92 elements of 'H')

```

activation internal region, example program

```

01771  P1  ptr/20145

```

The interpretation of a locator-qualified reference is now given in detail. Consider the reference:

P1->i(2*k-4) (assume k = 3, n = 2)

The steps in the interpretation are:

1. (Name Resolution.) The declaration of the major name, 'i', in the base reference is determined. It is given by the third 'declare' statement in the example program.
2. (Locator-Qualifier Evaluation.) The locator qualifier is the simple variable reference, 'P1'; its value is the base pointer value, '20145'. The associated storage type of the value must be the same as that declared for 'i'. In fact, the only valid source for such a value would be 'addr(H(8).item)'.
3. (Base-Variable Location.) The base pointer value is used to locate a position in storage. Three variables begin at this position: 'H(8).item(1).text' (a scalar), 'H(8).item(1)' (a structure), and 'H(8).item' (an array). The last of these has the same storage type as 'i', and it is therefore the base variable.

4. (Based-Variable Overlay.) The based variable is overlaid on the base variable. New designators are defined, and a portion of the diagram of storage is changed to read as follows:

```

20144          H (8).flags  "1 1 1/0/0/0"b
20145  i(1).t  ----  .item(1).text  "x x x x x x/0/n/a/i/i/s"
20147          .c  -----  .count  S 9 9 9 9 9/+/0/0/1/4/4
20151  (2).t  -----  (2).text  "a/C/C/i/d/e"
20153          .c  -----  .count  S 9 9 9 9 9/+/0/0/0/9/0
20155          .style  S 9 9 9 9/-/0/0/1/3

```

Observe that the variable 'i' matches the base variable, only because 'n' is '2' at the time this step is performed; if 'n' had any other value, the reference would be invalid.

5. (Reference Evaluation.) The subscript in the based reference is evaluated and the fully-bound reference is 'i(2)'. The result is:

```

storage type:  01,
                02 char(6),
                02 dec(5)

value:         "aCCide", 90

```

6. (Based-Variable Withdrawal.) The designators defined in Step 4 are discarded, and the diagram returns to the form that appeared at the beginning of this discussion.

Shortened Forms of References

The conventions for shortening variable references are described here, and specific guidelines for their use are given. A reference should be shortened only in order to make the program in which it occurs more clear to those who must read it. A reference should not be shortened merely to reduce the number of keystrokes required to type the program.

SUBSCRIPT-LIST DELETION

A subscripted reference can be shortened by deleting its subscript list, provided that all subscripts are '*' subscripts. Similarly, a structure-qualified reference can be shortened by deleting all of its subscript lists provided all of its subscripts are '*' subscripts.

Examples of Subscript List Deletion

Examples of the shortening of a reference by deletion of subscript lists are:

<u>Reference</u>	<u>Shortened Reference</u>
a(*)	a
b(*,*)	b
c(*) .d.e(*,*)	c.d.e

Observe that the third reference, 'c(*) .d.e(*,*)', cannot be shortened to 'c(*) .d.e' or 'c.d.e(*,*)'; if any subscript list is deleted, all must be.

Guidelines for Subscript List Deletion

The deletion of a subscript list is not recommended. Any reference with a '*' is necessarily a reference to an aggregate value. The PL/I facilities for handling aggregates are expensive and should not be used casually. Indeed, the presence of a subscript list composed of asterisks is a useful warning that an aggregate value is being processed.

NAME DELETION

A structure-qualified reference can be shortened by deleting one or more of its containing references, provided that the deleted references are unsubscripted names and provided that the deletion does not change the declaration of the reference.

The declaration of a reference is changed if the unshortened reference is governed by one declaration and the shortened reference is governed by another. The declaration that governs the reference is determined by rules given earlier in Section VI, "Declarations."

A subscript list can be moved within a reference. The purpose of this convention is to allow deletion of a containing reference which, aside from its subscript list, satisfies the conditions for deletion. For example, the reference 'x(i+2).y' can be written as 'x.y(i+2)' and then, if the declaration does not change, as 'y(i+2)'.

Examples of Name Deletion

The following program is used to illustrate the deletion of containing names to produce shortened references:

```
P:  proc;
    dcl 01 vehicle,
        02 serial dec(10),
        02 cost dec(8,2);
    ...
    begin;
        dcl 01 sale,
            02 customer,
            03 name char(30),
            03 address char(60),
            02 supplier,
            03 name char(8),
            03 cost(2) dec(10,2);
        dcl cost dec(10,2);
        ...
        ... (example references occur here)
        ...
    end;
end;
```

Observe that 'name' is declared twice and 'cost' is declared three times. These declarations are all valid but they make certain shortened references invalid.

Assume that the declarations explicitly shown are the only declarations in the program. Then the references that could be used in the inner block are classified as follows:

<u>Unshortened References</u>	<u>Valid Shortening</u>	<u>Invalid Shortening</u>
vehicle.serial	serial	
vehicle.cost		cost
sale.customer	customer	
sale.customer.name	customer.name	sale.name name
sale.customer.address	customer.address sale.address address	
sale.supplier.name	supplier.name	sale.name name
sale.supplier.cost(i)	supplier.cost(i) sale.cost(i)	cost(i)

The interesting references are the "invalid shortenings". They are accounted for as follows:

- The references 'sale.name' and 'name' are invalid because they are ambiguous. Each is a partially-qualified reference to both 'sale.customer.name' and 'sale.supplier.name'.
- The reference 'cost' is an invalid shortening of 'vehicle.cost' because (in the inner block) it would be interpreted as a reference to the 'cost' declared in the third 'declare' statement.
- The reference 'cost(i)' is an invalid shortening of 'sale.supplier.cost(i)' because (since subscripts are ignored in the resolution of a reference) it would also be interpreted as a reference to the 'cost' declared in the third 'declare' statement.

Guidelines for Name Deletion

The deletion of the leftmost reference of a structure-qualified variable reference is not recommended, even if a careful analysis shows that the result is correct. The leftmost level reference is much more important than the other level references because it determines which major variable is being referenced.

The moving of a subscript list from one level reference to another within a structure-qualified reference is not recommended. Although it has no effect on the interpretation of the reference by the processor, it gives the human reader the wrong storage type for the reference. The deletion of containing references should be confined to those that are originally unsubscripted.

LOCATOR QUALIFIER DELETION

A locator-qualified reference can be shortened by deleting the locator qualifier, provided that the major name of the based reference is declared with the attribute:

based(x)

where x is the locator qualifier that occurs in the unshortened reference.

Guidelines for Locator-Qualifier Deletion

The deletion of a locator-qualifier is not recommended for most applications. The use of based variables is an error-prone aspect of PL/I programming, and the deletion of a locator-qualifier introduces additional possibilities for errors.

Cost of Variable References

The relative complexity of the various kinds of variable references is a bad guide to the cost of these references. A PL/I program is compiled, and the cost of the interpretation of any construct, and references in particular, is divided between compilation and execution. Whatever can be performed during compilation becomes a negligible cost because it is performed only once for each compilation of the program.

The cost of interpreting a reference consists of a cost that is approximately the same for all variable references plus the cost of evaluating any expressions (subscripts or a locator qualifier) that occur in the reference. A long and complicated reference that contains only constant expressions, such as:

```
alpha(3).beta.gamma(5,2)
```

costs no more to interpret than a simple variable reference. Generally speaking, the organization of data into structures and the corresponding use of structure-qualified variables do not, in themselves, increase the cost of referencing the data.

CONSTANT LITERALS

A constant literal designates a computational constant value. The constant literal gives, in the spelling of the construct itself, both the data type and the value of the constant it designates. For example, the constant literal '23.9' designates a constant whose storage type is 'real fixed decimal (3,1)' and whose value is 23.9. The spelling of a constant literal is almost, but not quite, the same as the stored value representation it designates. For example, the constant literal '23.9' has the stored value representation '+23.9'.

A constant literal designates a constant that requires at most 64 words of storage (for the longest possible character-string constant). Furthermore, the value of a constant literal can always be determined at compilation time, and various optimization techniques can be applied to reduce the cost of its storage. For these reasons, the allocation and initialization of storage for the value designated by a constant literal need not be described. It is sufficient to show how, for a given constant literal, the storage type and value of the literal can be determined.

Arithmetic Constant Literals

The language provides a full range of arithmetic constant literals, including both 'fixed' and 'float' scaling, 'decimal' and 'binary' base, and 'real' and 'complex' mode.

FORM OF ARITHMETIC CONSTANT LITERALS

The form of the arithmetic constant literals is given by the following rules:

1. An integer literal is a sequence of one or more digits.
2. A fixed literal is either an integer literal or is an integer literal modified by the insertion of a decimal point before or after any digit.
3. A float literal is a fixed literal (called the mantissa) followed by an 'e' followed by a signed integer literal (called the exponent).
4. A decimal literal is any fixed or float literal. The exponent of a float decimal is considered to be a power of ten.
5. A binary literal is a fixed or float literal followed by a 'b'. Except for an exponent, the literal must contain only binary digits. The exponent of a float binary literal is interpreted as a decimal number; it is considered to be a power of two.
6. A real literal is any decimal or binary literal.
7. An imaginary literal is any decimal or binary literal followed by an 'i'.
8. An arithmetic literal is any real or imaginary literal.

An arithmetic literal must not be more than 256 characters long.

The rules just given build on one another. An integer literal is used to build a fixed literal, a fixed literal is used to build a float literal, and so on. The following shows how the rules are used to build the literals '11101110b' and 8.2300e-3i':

1.	integer	11101110	82300
2.	fixed	11101110	8.2300
3.	float	---	8.2300e-3
4.	decimal	---	8.2300e-3
5.	binary	11101110b	---
6.	real	11101110b	8.2300e-3
7.	imaginary	---	8.2300e-3i
8.	arithmetic	11101110b	8.2300e-3i

INTERPRETATION OF ARITHMETIC CONSTANT LITERALS

The interpretation of an arithmetic constant literal must yield a storage type and a value. The storage type is determined as follows:

- The aggregate type is always scalar.
- The mode is 'complex' if the reference ends with an 'i', and is 'real' otherwise. Default statements can not change the mode of a constant.
- The scaling is 'float' if the reference has an 'e' followed by an exponent and is 'fixed' otherwise.
- The base is 'binary' if the reference contains a 'b' and is 'decimal' otherwise.
- The number-of-digits in the precision is obtained by counting all the digits except those in exponents 'e' or 'f'.
- The scale-factor in the precision is obtained by counting the digits to the right of the point except for those in an exponent. If there is no point, the scale factor is zero.
- If the arithmetic constant contains a 'p', default processing of that constant does not take place; 'p' is called the default suppression character.

The value of the constant literal is the value represented by the spelling of the literal.

EXAMPLES OF ARITHMETIC CONSTANT LITERALS

Examples of arithmetic constant literals follow. Each example is accompanied by its storage type and its representation in storage.

<u>Constant Literal</u>	<u>Storage Type</u>	<u>Representation</u>
23.9	real fixed decimal(3,1)	+23.9
0	real fixed decimal(1,0)	+0.
000	real fixed decimal(3,0)	+000.
110.011011101b	real fixed binary(12,9)	+110.011011101b
0b	real fixed binary(1,0)	+0.b
5.8000000e3	real float decimal(8)	+58000000.e-4
.1110010011101e-3b	real float binary(13)	+.1110010011101e-3b
568e0i	complex float decimal(3)	+000.e0+568.e0i
9.000e5i	complex float decimal(4)	+0000.e0+9000.e2i
10101bi	complex fixed binary(5,0)	+00000.b+10101.bi

If 2 is specified, the character(s) are restricted to 0, 1, 2, and 3. If 3 is specified, the character(s) are restricted to all digits except 8 and 9. If 4 is specified, the character(s) are restricted to 'digit', a, b, c, d, e, and f. The number, if any, after the "b" indicates how many bits each digit represents. For hexadecimal, uppercase or lowercase may be used but they cannot be intermixed.

3. A replication factor is a parenthesized integer literal whose value is greater than zero. Suppose the value of the replication factor for a given reference is n and that the sequence of characters between the double-quote characters is s. Then an equivalent string literal is obtained by replacing s with n copies of s and deleting the replication factor.

A string literal must not be more than 256 characters long. If a string literal has a replication factor, the restriction on length is applied to the equivalent literal that does not have a replication factor. For example, '(254)"0"b' is considered to be a literal of 257 characters and therefore is invalid.

INTERPRETATION OF STRING CONSTANT LITERALS

The interpretation of a string constant literal must yield a storage type and a value. The storage type is determined as follows:

- The aggregate type is always scalar.
- The literal is 'bit(n)' or 'character(n)' depending on whether a 'b' occurs at the end or not. The n is the number of characters in the sequence between the double-quote characters; a pair of double-quote characters in the sequence counts as one character.
- The literal is always 'nonvarying'.

The value of the literal is the sequence between double-quote characters with the provision, already noted, that two double-quote characters in the sequence represent one double-quote character in the value.

EXAMPLES OF STRING CONSTANT LITERALS

Examples of string constant literals follow:

<u>Constant Literal</u>	<u>Storage Type</u>	<u>Value Representation</u>
"Say nothing."	character(12) nonvarying	"Say nothing."
"Say ""Stop ""."	character(12) nonvarying	"Say "Stop "."
""	character(0) nonvarying	""
"100011"b	bit(6) nonvarying	"100011"b
"1"b	bit(1) nonvarying	"1"b
""b	bit(0) nonvarying	""b
(2)"moshe "	character(12) nonvarying	"moshe moshe "
(12)"1"b	bit(12) nonvarying	"111111111111"b

ESCAPE CONVENTIONS FOR CHARACTERS

For terminals that do not provide the full ASCII character set, escape conventions must be used to type in certain characters when they are required in a character-string constant literal. These conventions are part of Multics and are not peculiar to PL/I. They are mentioned only briefly here:

- For a character that is available on the terminal in use, type the key or key combination for that character.
- For a character that is not available but that has a special multi-character equivalent for the terminal in use, type the equivalent.
- For a character that is not available and for which the programmer does not know a special equivalent, type a backslash character followed by three digits which are an octal representation of the ASCII code for the character.

The backslash and octal code combination is a universal escape convention that applies throughout Multics; it is available for use under any circumstances. The special multi-character equivalents are more concise, but they vary from one terminal to another; they are given in the Multics Programmer's Manual.

As an example of the escape conventions, suppose that a programmer is typing in a program at a Model 33 Teletype and wants to enter the statement

```
mes = "{Start}";
```

The Model 33 has one case of letters, 'A' through 'Z', and these letters are used in Multics as if they were lower case. Therefore, the problem letters in this example are the left brace, the capital 'S', and the right brace. If the programmer does not know the special escape conventions for the Model 33, he can use the universal escape convention and type:

```
MES = "\173\123TART\175";
```

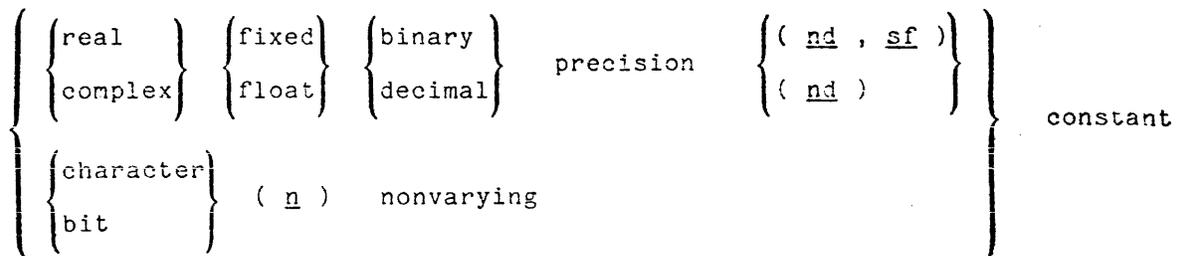
On the other hand, if the programmer knows the special conventions for the Model 33, he can type the statement more concisely as:

```
MES = "\(\START\)";
```

If the resulting input is later typed out on the Model 33 in "edited" mode, the statement will appear in the form just given regardless of how it was typed in.

Attributes for Constant Literals

The complete attribute set for any constant literal is given by the following diagram:



In this diagram,

- nd (number of digits) is an unsigned integer
- sf (scale factor) is an optionally signed integer
- n (maximum length) is an unsigned integer

The attribute set for a constant literal is never written in a program; but it must be determined as part of the interpretation of the program.

CONSTANT REFERENCES

A constant reference designates a constant statement address value or a constant 'file' value. A constant reference can appear in two kinds of context. The most common context is one that makes final use of the value of the constant reference; for example, a 'label' constant reference in a 'goto' statement provides the destination for transfer of control. The second context is one that saves the value of the constant reference for later use; for example, the assignment of the value of a 'label' constant reference to a 'label' variable or the use of a 'label' constant reference as an argument in a function reference.

Observe that the range of constant references is limited. Except for 'label' constant references, they handle only scalar values. No provision is made for constant references for 'pointer', 'offset', or 'area' values.

Statement Constant References

A statement constant reference can appear wherever a statement address value is required. The most common use of a statement constant reference is in a context that makes final use of the value of the reference. For a 'label' constant reference, this context is a 'goto' statement; for an 'entry' constant reference, it is a 'call' statement or a function reference; and for a 'format' statement, it is the 'r' format item that is used in connection with edit-directed stream input/output.

FORM OF STATEMENT CONSTANT REFERENCES

A statement constant reference has one of the following forms:

scn

scn(se)

where scn is the statement constant name and se is the subscript expressions. The statement constant name must be an identifier that has the type attribute 'label', 'entry', or 'format'. The second form can be used only with a 'label' constant name. The subscript expression must yield a value that can be converted to an integer.

INTERPRETATION OF STATEMENT CONSTANT REFERENCES

The evaluation of a statement constant reference yields a 'label' value, an 'entry' value, or a 'format' value. Such a value contains a statement designator and an activation index. The activation index has no significance except in a program that uses general recursion; it is described later, at the end of Section XII, "Procedure Invocation."

The statement designator that is part of the value of a statement constant reference is defined by the label prefix that declares the identifier in the statement constant reference. For example, the statement constant reference 'alpha' designates a statement that (1) has the label prefix 'alpha:' and (2) is contained in the smallest block that contains both the given reference and a statement with the label prefix 'alpha:'.

EXAMPLES OF STATEMENT CONSTANT REFERENCES

As a simple example of the use of a statement constant reference, consider the following program:

```
P1: proc;
LAB:  call SR(x);
      ...
      goto LAB;
      ...
      end;
```

The occurrence of 'LAB:' at the beginning of the 'call' statement is the defining label prefix for 'LAB'. By virtue of that prefix, 'LAB' is declared 'label internal' and is given the address of the 'call' statement as its value. The occurrence of 'LAB' in the 'goto' statement is a 'label' constant reference.

As a more general example of the use of statement address constant references, consider:

```
P2: proc;
      ... (Computation #1)
      call M3;
      put edit( ... )(r(F));
M3:  proc;
      ... (Computation #2)
      goto L(i+2);
L(1): ... (Computation #3)
      goto A;
L(-1): ... (Computation #4)
      goto A;
L(2):
A:    ... (Computation #5)
      end;
F:    format( ... );
      end;
```

In the 'call' statement, 'M3' is an 'entry' constant reference; in the 'put' statement, 'F' is a 'format' constant reference; and in the 'goto' statement, 'L(i+2)' is a 'label' constant reference.

The example program performs Computation #1, calls the procedure 'M3', and executes the 'put' statement. When the procedure 'M3' is called, it performs Computation #2 and then proceeds as follows:

- If $i+2=1$, it performs Computation #3 and Computation #5.
- If $i+2=-1$, it performs Computation #4 and Computation #5.
- If $i+2=2$, it performs Computation #5.
- If $i+2=0$, execution is undefined.
- If $i+2$ is not in the range -1 through 2 , the 'subscriptrange' condition occurs.

When the 'put' statement is executed, a reference is made to the 'format' statement, and that statement supplies the format items for the output.

EXTERNAL 'entry' CONSTANT REFERENCES

There is one case in which a label prefix is not sufficient declaration for a statement constant name; this case arises when an external 'entry' constant is defined in one external procedure and is used in another external procedure. In the defining procedure, the constant is declared by a label prefix as already described. However, in other external procedures that refer to the 'entry' constant, the constant name must be declared again; that is, the name must be declared with the 'external' and 'entry' attributes by means of a 'declare' statement.

As an example of the declaration of external 'entry' constant names, consider the following program:

```
P1:  proc;
      dcl P2 entry(float,dec(10));
      dcl P3 entry(float);
      ...
      call P2(x,y);
      ...
      call P3(z);
      ...
      end;

P2:  proc(R,S);
      dcl R float;
      dcl S dec(10);
      ...
P3:  entry(Q);
      dcl Q float;
      ...
      end;
```

This program is made up of two external procedures. The declarations of 'P2' and 'P3' in the second external procedure are provided by label prefixes. Because these names are used in the first external procedure, they must be declared in that procedure by means of 'declare' statements.

File Constant References

A file constant reference can appear wherever a file value is required. The most common use of a file constant reference is in the context that makes final use of the value of the reference; that is, a 'file' option in a statement that performs input/output or opens or closes a file.

FORM OF FILE CONSTANT REFERENCES

A file constant reference has the following form:

fcn

where fcn is a file constant name. A file constant name is an identifier that is declared with the attribute 'file'.

EXAMPLES OF FILE CONSTANT REFERENCES

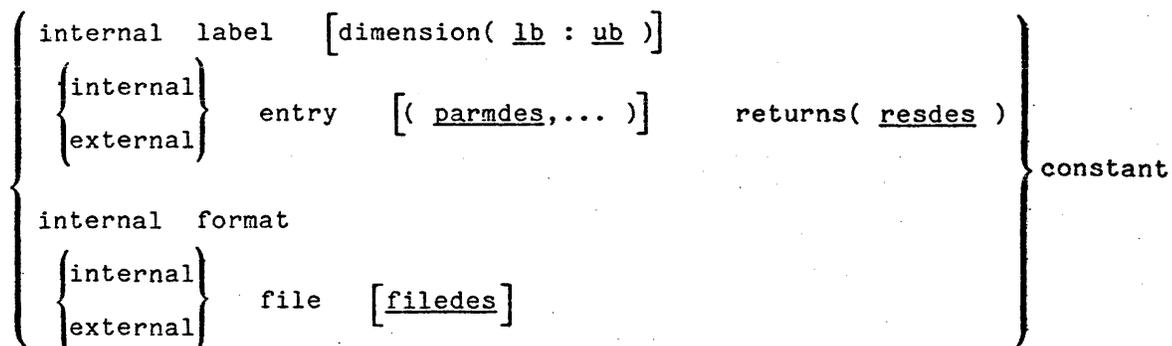
As an example of the use of file constant references, consider the following program:

```
P:  proc;
      dcl (alpha,beta) file;
      ...
      open file(alpha) input stream;
      open file(beta) output print stream;
      ...
      get file(alpha) list(a,b,c);
      ...
      put file(beta) list(x,y,z);
      ...
      close file(alpha);
      close file(beta);
      ...
      end;
```

This program shows how the file constant references 'alpha' and 'beta' are used to designate file-state blocks for an input data set and an output data set, respectively.

Attributes for Constant Names

The complete attribute set for named constants is given by the following diagram:



In this diagram,

- lb (lower bound) and ub (upper bound) are optionally-signed integers.
- parmdes is a parameter descriptor and resdes is a result descriptor; these constructs are described later, in Section XII, "Procedure Invocation."
- filedes is a file description; this construct is described later, in Sections XIV and XV, "Stream Input/Output" and "Record Input/Output," respectively.

The attribute set for a 'label' or 'format' constant name is never written in a program; instead, it is deduced from a label prefix. The attribute set for an entry constant name is written in a 'declare' statement only in an external procedure that uses but does not define the entry constant; otherwise, it is deduced from a label prefix. The attribute set for a file constant name is always given in a 'declare' statement.

PROGRAMMED FUNCTION REFERENCES

A programmed function reference invokes a PL/I procedure and then delivers the result of the execution of the procedure as the value of the reference. Through the use of a programmed function reference, a long and complicated procedure can be executed in the midst of the evaluation of an expression.

Form of Programmed Function References

A programmed function reference has one of the following forms:

ref(arglist)

ref()

where ref is the entry reference and arglist is the argument list. Usually, the entry reference is an entry constant name; however, it can also be a generic entry name or any reference that yields a scalar entry value. The argument list is a sequence of arguments separated by commas, and each argument is an expression. The second form is used when no arguments are required.

A generic entry name does not directly designate an 'entry' value; instead, it is replaced by an entry constant name during the compilation of the program. The declaration of the generic entry name gives a set of entry name constants. For each entry name constant in the declaration, some attributes for each argument of the designated procedure entry point are given. Thus when a programmed function reference begins with a generic function name, the interpretation of the generic entry name is determined by the storage types of the arguments of the programmed function reference. For a description of the individual programmed function references, see the PL/I Language Specification. An example of the use of a generic entry name is given in Section XII, "Procedure Invocation."

Interpretation of Programmed Function References

The interpretation of programmed function references is fully described later, in Section XII, "Procedure Invocation." That interpretation can be summarized in two steps, as follows:

1. Storage Type Determination. Obtain the storage type of the entry reference of the given programmed function reference. This storage type includes an 'entry' attribute and a 'return' attribute, and these attributes provide information about the procedure entry point that is designated by the entry reference. The 'entry' attribute gives a storage type for each parameter of the designated procedure entry point; this information is used in interpreting the argument. The 'returns' attribute gives the storage type of the value returned by the designated entry point and thus gives the storage type of the given programmed function reference.
2. Reference Evaluation. Determine the value of the programmed function reference. This requires the evaluation of the entry reference; the interpretation of the arguments; the activation, execution, and exit from the procedure; and the retrieval of the result of the procedure.

Step 1 of this interpretation is performed by the compiler before the program is executed. Step 2 is performed each time the programmed function reference is evaluated during program execution.

Examples of Programmed Function References

As a simple example of the use of programmed function references, consider the following program:

```
P:  proc;
    dcl (sysin,sysprint) file;
    dcl (a,b,c) float;
    get list(a,b);
    c = F(a) + F(b);
    put list(c);
F:  proc(x) returns(float);
    dcl x float;
    if x < 0
        then return(0);
        else return(x);
    end;
end;
```

In this program, the procedure 'F' is especially simple; its result is either zero or the given argument value depending on whether the argument is negative or positive. The important point, however, is that the definition of 'F' is a separate part of the program and can be examined and modified separately.

In order to execute the example program, it is necessary to interpret the following operator expression:

F(a) + F(b)

In order to interpret this expression, it is necessary to know the storage type of the two programmed function references that appear in it. Consider the way in which the storage type of 'F(a)' is determined. First, the storage type of the entry reference, 'F', is determined; it is:

entry(float) returns(float)

This storage type is obtained by applying the rules for the interpretation of an 'entry' constant reference to 'F'; those rules are given earlier in this section. The storage type of 'F(a)' is obtained from the 'returns' attribute in the storage type of 'F'; it is:

float

In the same way, it can be shown that the storage type of 'F(b)' is also 'float'.

A second example of a programmed function reference follows. This example uses both a variable reference and a programmed function reference as the entry reference of a programmed function reference. Such usage occurs only in large and complicated programs, and a short example cannot be realistic; however, the example is formally correct and is used to show how the storage type of a complicated entry reference is determined.

```
P:  proc;
    dcl (sysin,sysprint) file;
    dcl (m,n) fixed;
    dcl x float;
    dcl fv entry(fixed) returns(entry(float) returns(float)) variable;
    get list(m,n,x);
    if m=0 then fv = F1; else fv = F2;
    put list(fv(n)(x));
F1:  proc(a) returns(entry(float) returns(float));
    dcl a fixed;
    if a=0 then return(F3); else return(F4);
    end;
F2:  proc(b) returns(entry(float) returns(float));
    dcl b fixed;
    if b=0 then return(F4); else return(F3);
    end;
F3:  proc(z1) returns(float);
    dcl z1 float;
    return(sin(z1));
    end;
F4:  proc (z2) returns (float);
    dcl z2 float;
    return(sin(z2));
    end;
end;
```

The central feature of the program is the interpretation of the programmed function reference:

`fv(n)(x)`

By the time this reference is evaluated, one of the 'entry' constant references 'F1' or 'F2' has been assigned to the 'entry' variable named 'fv'. Thus the given reference is equivalent to one of the following:

`F1(n)(x)` (for $m = 0$)

`F2(n)(x)` (for $m \neq 0$)

The evaluation of the programmed function reference 'F1(n)' yields the 'entry' value designated by either 'F3' or 'F4', depending on whether 'n' is zero or not. Similarly, the evaluation of 'F2(n)' yields the 'entry' value designated by either 'F4' or 'F3', depending on whether 'n' is zero or not. Thus, finally, the given reference is equivalent to one of the following:

`F3(x)` (for m and n both zero or both nonzero)

`F4(x)` (for other values of m and n)

Clearly this example could be programmed in a simpler and more efficient way, but it would not then illustrate the use of a nonconstant entry reference.

The storage type of 'fv(n)(x)' can be obtained before program execution begins, as follows. First, the storage type of the variable 'fv' is determined from the 'declare' statement to be:

`entry(fixed) returns(entry(float) returns(float))`

This storage type means that the value of 'fv' is "an 'entry' value that designates a procedure entry point that has a 'fixed' parameter and returns an 'entry' value; and the latter 'entry' value designates a procedure entry point that has a 'float' parameter and returns a 'float' value". Next, the storage type of the programmed function reference ('fv(n)') is determined from the storage type of 'fv' to be:

`entry(float) returns(float)`

This storage type means that the value of 'fv(n)' is "an 'entry' value that designates a procedure entry point that has a 'float' parameter and returns a 'float' value". Finally, the storage type of the entire reference is determined from the storage type of 'fv(n)' to be:

`float`

This storage type means, of course, that the value of 'fv(n)(x)' is "a 'float' value".

BUILT-IN FUNCTION REFERENCES

A built-in function reference performs a specific calculation on its arguments and delivers the result as the value of the reference. For each built-in function, the calculation performed is part of the definition of PL/I. Although a built-in function reference resembles a programmed function reference in some ways, there are important and fundamental differences between the two kinds of reference. These differences are discussed here, after the form and interpretation of built-in function references are given.

A specific definition for each of the built-in functions is given in Section IX, "Operations." Each definition gives restrictions on the arguments and gives rules for converting the arguments, determining the storage type of the result, and calculating the value of the result. The section on "Operations" is large, and the quickest way to find the definition of a particular built-in function is to look its name up in the index of this manual.

Form of Built-In Function References

A built-in function reference has one of the following forms:

bifname(arglist)

bifname()

bifname

where bifname is the built-in function name and arglist is the argument list. The built-in function name must be declared 'builtin' and must be one of the following identifiers:

abs	ceil	dimension	lineno	onloc	sqrt
acos	character	dim	log	onsource	stac
add	char	divide	log10	pageno	stackbaseptr
addr	clock	dot	log2	pointer	stackframeptr
addrel	codeptr	empty	low	ptr	stacq
after	collate	environmentptr	itrim	precision	string
allocation	collate9	erf	max	prec	substr
allocn	complex	erfc	maxlength	proc	subtract
asin	cplx	exp	min	real	sum
atan	conjg	fixed	mod	rel	tan
atand	convert	float	multiply	reverse	tand
atanh	copy	floor	null	round	tanh
baseno	cos	hbound	nullo	rtrim	time
baseptr	cosd	high	offset	search	translate
before	cosh	high9	onchar	sign	trunc
binary	currentsize	imag	oncode	sin	unspec
bin	date	index	onfield	sind	valid
bit	decat	lbound	onfile	sinh	vclock
bool	decimal	length	onkey	size	verify
	dec				

The argument list is a sequence of expressions separated by commas. The second and third forms of a built-in function reference are equivalent; either can be used for a built-in function that requires no arguments.

Interpretation of Built-in Function References

A programmed function reference is interpreted in two steps, as follows:

1. Storage Type Determination. Determine the target storage type for each argument and the storage type of the result. The rules for determining these storage types are given in the individual definitions of the built-in functions; often they depend on the storage types of the arguments.
2. Reference Evaluation. Evaluate each argument and convert it to the target storage type for the argument. Evaluate the reference as specified in the definition of the built-in function.

Step 1 of this interpretation is performed by the compiler before the program is executed. Step 2 is performed each time the built-in function reference is evaluated during program execution.

Examples of Built-in Function References

As an example of a built-in function reference that handles storage types in a simple but quite typical way, consider the following use of the 'sin' built-in function:

```
dcl (y,alpha) float;
...
y = sin(alpha);
```

The function reference under consideration is:

```
sin(alpha)
```

The built-in function name is 'sin' and the argument list contains one argument, 'alpha'.

The storage type of the argument, 'alpha', of the built-in function reference in the example is:

```
real float binary(27)
```

According to the definition of 'sin' in Section IX, "Operations," the target storage type is:

```
real float binary(27)
```

Thus for this use of 'sin', the argument is not converted before the calculation begins. Also according to the definition of 'sin', the result storage type is:

```
real float binary(27)
```

Thus the storage type of the result is the same as that of the argument. When the assignment statement is executed, the value of 'alpha' is fetched, the sine is calculated, and the result is returned as the value of the reference.

As an example of a more complicated built-in function reference, consider the following use of the 'max' built-in function:

```
dcl x float;
dcl a float;
dcl b fixed(35);
...
x = max(a,b,200);
```

Here, the assignment statement assigns the largest of the values designated by 'a', 'b', and '200' to the variable named 'x'. A precise understanding of the assignment statement requires the determination of the storage type of the result of the reference to 'max'.

The following table gives the argument storage types and the target storage types for the built-in function reference:

<u>Argument</u>	<u>Argument Storage Type</u>	<u>Target Storage Type</u>
a	real fixed binary(27)	real float binary(27)
b	real fixed binary(35)	real float binary(35)
200	real fixed decimal(3)	real float binary(10)

Observe that the target storage types differ only in the precision attribute; the choice of 'float' over 'fixed' and of 'binary' over 'decimal' is a rule that applies to many built-in functions. The storage type of the result is:

```
real float binary(35)
```

Observe that the number-of-digits is the maximum of those given in the target storage types of the arguments.

As an example of a built-in function reference with aggregate arguments, consider the following use of the 'max' and 'min' built-in functions:

```
dcl 01 alpha(3),
    02 a float,
    02 b fixed;
dcl 01 top,
    02 a float,
    02 b fixed;
...
alpha = min(max(0,alpha),top);
```

Here, the assignment statement changes the value of 'alpha' where necessary so that its components lie in the range:

$$0 \leq \text{alpha.a}(i) \leq \text{top.a} \quad (\text{for } i = 1, 2, \text{ and } 3)$$

$$0 \leq \text{alpha.b}(i) \leq \text{top.b}$$

When a value is changed, it is changed as little as possible; for example, a negative value is changed to zero.

The following table gives the argument storage types and the target storage types for the built-in function references:

<u>Argument</u>	<u>Argument Storage Type</u>	<u>Target Storage Type</u>
0	real fixed dec(1)	01 dim(1:3), 02 real float bin(4), 02 real fixed bin(4)
alpha	01 dim(1:3), 02 real float bin(27), 02 real fixed bin(17)	01 dim(1:3), 02 real float bin(27), 02 real fixed bin(17)
top	01, 02 real float bin(27), 02 real fixed bin(17)	01 dim(1:3), 02 real float bin(27), 02 real fixed bin(17)

Observe that the scalar, '0', and the structure, 'top', are converted to the aggregate type of 'alpha'.

Differences Between Built-in and Programmed Function References

The essential difference between built-in and programmed function references is in the way the actions performed are defined. For a built-in function reference, the action is defined as part of the PL/I language, and a given built-in function name means the same thing wherever it is used. In contrast, for a programmed function reference, the action is defined as part of a program, and a given programmed function name can mean different things under different circumstances.

In addition to this essential difference, there are several other important differences between built-in and programmed function references. These differences are:

1. A built-in function reference must begin with a built-in function name; therefore, the selection of the built-in function is made when the program is written. In contrast, a programmed function reference can, when necessary, begin with an entry variable reference or an entry programmed function reference; therefore, the selection of the programmed function can be made as part of each evaluation of the programmed function reference, and can change from one evaluation to the next.
2. A specific argument of a specific built-in function reference can have any of several storage types without undergoing conversion of its storage type. In contrast, unless a generic function name is used, an argument of a programmed function reference must have a specific storage type (except for variations in extents) in order to escape conversion.
3. The result of a built-in function reference has a storage type that is derived from the storage type of its arguments. In contrast, the result of a programmed function reference is independent of the storage types of its arguments (except, perhaps for extents) and is determined by the definition of the function.
4. On a less important level, the parentheses around the argument list can be omitted from a built-in function reference that has no arguments. In contrast, the parentheses must be given with a programmed function reference even if there are no arguments.
5. Finally, a built-in function reference is evaluated at a relatively low cost. In contrast, the evaluation of a programmed function reference is relatively expensive, even when the invoked procedure consists of only a few simple statements.

This list shows that the difference between built-in and programmed function references are more important than the similarities.

OPERATOR EXPRESSIONS

Like a built-in function reference, an operator expression performs a specific calculation on its arguments and delivers the result as the value of the expression. For each operator, the calculation performed is part of the definition of PL/I. The only difference between operators and built-in function references is a difference in form. An operator expression can be thought of as a built-in operation that, because of its frequency of use, is represented by means of a special notation, using an operator, rather than by means of the less compact built-in function reference.

A specific definition of each of the operators is given in Section IX, "Operations." Just as for built-in functions, each definition of an operator gives restrictions on the arguments and gives rules for calculating the value of the result. A quick way to find the definition of a particular operator is to look the operator up in the index of this manual.

Form of Operator Expressions

An operator expression has one of the following forms:

```
( arg1 inop arg2 )      arg1 inop arg2
( preop arg1 )          preop arg1
```

where arg1 and arg2 are the operands (also called arguments) of the operator, inop is the infix operator, and preop is the prefix operator. An infix operator must be one of the following:

```
** * / + - ||
= ^= < ^< > ^> <= >=
& |
```

A prefix operator must be one of the following:

```
+ - ^
```

For both the infix operator and prefix operator a parenthesized and an unparenthesized form is given. It is always correct to use the parenthesized form of an operator expression. The unparenthesized form can be used when the priority rules, described later in this discussion of operator expressions, provide the interpretation that the programmer wants.

Interpretation of Operator Expressions

An operator expression is interpreted in the same two steps that apply to a built-in function reference, as follows:

1. Storage Type Determination. Determine the target storage type for each operand and the storage type of the result. The rules for determining these storage types are given in the individual definitions of the operators; often they depend on the storage types of the operands.
2. Reference Evaluation. Evaluate each operand and convert it to the target storage type for the operand. Evaluate the reference as specified in the definition of the operator.

Step 1 of this interpretation is performed by the compiler before the program is executed. Step 2 is performed each time the operator expression is evaluated during program execution.

Operator Priority Rules

Two operators are on the same expression level if they appear in the same expression and the only parentheses that appear between them are matched pairs. When there are several operators on the same expression level, the following table determines the order in which the operators are evaluated:

<u>Priority</u>	<u>Operators</u>	<u>Order within Priority</u>
highest	** ^ prefix + prefix - * / infix + infix - 	right to left
	= ^= < ^< > ^> <= >= &	left to right
lowest		

When two operators appear on the same expression level, the operator with higher priority is evaluated first. If the operators have the same priority, then they are evaluated in left to right or in right to left order, depending on the entry in the third column of the table. These are the operator priority rules.

As an example of the application of the operator priority rules, consider the following expression:

$$4*(a-(b/c)**2+d)$$

There are three expression levels in this expression. One of them has just one operator, '*', and another also has just one operator, '/'. However, the remaining expression level contains three operators, '-', '**', and '+'. The rules are applied as follows:

- The operator with the highest priority is '**'; therefore, this operator is evaluated before the other operators on the same expression level.
- The remaining two operators, '-' and '+', both have the same priority. According to the table, they are evaluated from left to right; therefore, '-' is evaluated before '+'.

According to this analysis, the example is equivalent to the following expression:

$$4*((a-((b/c)**2))+d)$$

This expression makes the required order of evaluation explicit.

In some cases, the results of the operator priority rules are consistent with well-known conventions of mathematical notation. In other cases, the results do not correspond in any obvious way to familiar notation. Some examples of the latter are:

<u>Given</u>	<u>Equivalent</u>
a/b/c	(a/b)/c
a**b**c	a**(b**c)
-a**b	-(a**b)
a*-b	a*(-b)
a=b=c	(a=b)=c

In each of these cases it is suggested that the given expression be avoided and that, instead, the parenthesized equivalent be used. Parentheses should not be omitted except where the interpretation will be obvious to everyone who must read the program.

The operator priority rules do not fully determine the order in which operators in an expression are evaluated; it applies only to operators that are on the same expression level. Consider, for example:

```
(a+b)*(c+d)
```

In this expression there are three expression levels, and each contains only one operator; therefore, the operator priority rules say nothing about this example. From the fact that the '+' operators are contained in the operands of the '*', it can be concluded that the '+' operators are evaluated first. However, the order in which the two '+' operators are evaluated is not defined.

Examples of Operator Expressions

As an example of the use of operator expressions, consider the following program fragment:

```

dcl (y,a,b,c,d) float;
...
y = a*(b-c)+d;

```

According to the priority rules, the right-hand side of the assignment statement is equivalent to:

```
(a*(b-c))+d
```

The right-hand side is made up of three operator expressions.

According to the definition of the '*', '-', and '+' operators, the target storage type for all these operands is:

```
real float binary(27)
```

When the definitions of the operators are applied to this expression, it turns out that the target storage type for every operand and the result storage type for every operator is also:

```
real float binary(27)
```

This simplicity with respect to storage types is typical of scalar, floating-point calculations.

As a more complicated example of the case of operator expressions, consider:

```
decl i fixed;  
...  
i = i+1;
```

In this example, the assignment statement increases the value of 'i' by one. The calculation is simple, but the determination of the storage type of the right-hand-side expressions is not.

The following table gives the operand storage types and the target storage types as determined by the definition of the '+' operator.

<u>Operand</u>	<u>Operand Storage Type</u>	<u>Target Storage Type</u>
i	real fixed binary(17)	real fixed binary(17)
1	real fixed decimal(1)	real fixed binary(4)

The storage type of the result is:

```
real fixed binary(18)
```

Observe that the precision of the target and result storage types accommodate any value of 'x' that can arise.

SECTION IX

OPERATIONS

The operations of PL/I are invoked by the operators and the built-in functions. It is the operations that determine the computational foundation of PL/I. There are well over a hundred operations, and they are described here under the following headings:

- Arithmetic Operations. These operations perform the fundamental operations of arithmetic. They range from such elementary operations as addition and subtraction to less well known operations such as the modulus and the conjugate.
- Mathematical Operations. These are the transcendental functions of applied mathematics, including the exponential and logarithmic functions and the standard trigonometric functions.
- String Operations. These operations manipulate string values. They range from the fundamental concatenate operator and 'substring' function to advanced functions for text processing.
- Address and Area Operations. These functions manipulate address values and area values. They are rarely used outside of advanced programming applications.
- Array Operations. These operations are especially designed to operate on array arguments.
- Conversion Operations. These operations are used to convert a value of one storage type to another.
- Special Operations. These operations are intimately related to the PL/I processor; they access system variables or depend on the details of program execution.

The definitions given here show the result of a given operation applied to given argument values. The definitions do not discuss the context of the operation. The way in which operations fit into the context of a program is described in Section VIII, "Expressions."

Following this introduction, this section gives general rules and conventions that apply to all of the definitions of operations. The section then continues through seven subsections, each of which begins by giving the conventions that apply to that subsection. In many cases, a complete understanding of a given operation requires familiarity with both the general conventions and the conventions for the subsection in which the given operation is defined.

GENERAL REMARKS

Remarks that apply to all of the definitions in this section are given here. They apply to argument evaluation, nonstandard operations, and the conventions for the presentation of examples.

Argument Evaluation

The evaluation of the arguments of an operator expression or a built-in function reference is described in detail in Section VIII, "Expressions." Three important points are stated briefly here:

- When an operator expression or a built-in function reference has several arguments, the order in which the arguments are evaluated is not defined. The processor may, for example, begin the evaluation of the first argument, perform the evaluation of the second argument, and then complete the evaluation of the first argument.
- Except where the definition of an operation includes a restriction to the contrary, one or more of the arguments can be an aggregate. If the arguments of the operation have different aggregate types, then at least one of the arguments must have an aggregate type that is a suitable target for the conversion of the other arguments; before the operation is performed, all the arguments are converted to this aggregate type according to the rules given earlier, in Section IV, "Value Conversion." When the arguments all have the same aggregate type (either by conversion or because they were given that way), the operation is applied to corresponding scalar components of the arguments just as it would be applied to scalar arguments. The result has the same aggregate type as the arguments. Examples of operations on aggregate arguments are given earlier, in Section VIII, "Expressions."
- Standard PL/I allows conversion between any computational data types. However, Multics, as a matter of policy, discourages implicit conversion between the following general types of values:

- arithmetic
 - character string
 - bit string

The Multics PL/I compiler prints a warning message whenever implicit conversion between these general types is required by context. Therefore, the definitions given in this section specify that a given argument must be one of these three major types. When it is necessary, for example, to use a character-string argument for an operation that requires an arithmetic value, one of the functions described under "Conversion Operations" in this section should be used to perform the conversion explicitly.

Non-Standard Operations

The Multics implementation of PL/I includes certain operations that are not part of Standard PL/I. The heading of the definition of each such operation ends with an asterisk(*) and the function is referred to as non-standard in its description. There are two kinds of nonstandard functions. The first kind is an extension of PL/I and is designed to make programming easier; the "Hyperbolic Functions" are examples. This kind of operation can be eliminated quite easily in the event a program must be reduced to Standard PL/I. The second kind of nonstandard function operation depends on the way Multics PL/I is implemented; the "Implementation-Dependent Address Functions" are examples. A program that uses these operations must be substantially revised to reduce it to the Standard.

Conventions for Examples

Many examples are given in this section. In each example, the values of the arguments and the result are given as constants whose data types are correct for the example. Consider the following example of the plus-sign operator:

$$(006.) + (-2.00) = 0004.00$$

This example not only shows that the value of six plus minus two is four, but also shows that a 'dec(3)' value plus a 'dec(3,2)' value yields a 'dec(6,2)' result. Often the value of the result is obvious and the purpose of the example is to show the storage type of the result.

Each example consists of an operator expression or a function reference, followed by an equals sign, followed by a result. (The example given in the previous paragraph has this form.) The equals sign means "yields the result"; it is not used as a PL/I symbol.

For certain argument values, the evaluation of an operator expression or a function reference causes a condition to occur; in that case, the condition name, enclosed in parentheses, is given instead of the result. An example is:

$$(2.0000e0) / (0.0000e0) = (\text{zerodivide})$$

This example shows that division by zero causes the 'zerodivide' condition to occur.

In many of the examples that have arithmetic arguments, 'decimal' rather than 'binary' arguments are shown. The 'decimal' base is used for these examples because it is easier to understand. The use of 'decimal' base is not an endorsement; in fact, 'decimal' values are rarely used in practice outside of business programming.

ARITHMETIC OPERATIONS

The arithmetic operations manipulate arithmetic values. A single operation, such as the '+' operator, can have many interpretations depending on the storage types of its arguments. The arithmetic operations contrast with the mathematical operations, discussed later, which perform transcendental operations and always produce floating-point results.

The rules for determining the data type of the result of an arithmetic operation are sometimes complicated. Much of the complexity arises in determining the scale-factor of a 'fixed' result. In many programming applications, it is possible to avoid 'fixed' data that has a nonzero scale factor; that is, fixed-point data can be confined to integers. In these applications, the rules become simpler.

Conventions for Definitions

The arithmetic operations defined here have some points in common, and certain conventions and general rules apply to all of the definitions.

ARGUMENTS

As the first step in the interpretation of an operation, the storage types are checked. Each arithmetic operation requires arguments that yield arithmetic values; in some cases, an argument is further required to be either 'real' or 'complex'. These requirements are indicated by the use of the following conventions:

X, X1, X2 ... The value of the argument must be arithmetic

R, R1, R2 ... The value of the argument must be 'real' arithmetic

Z, Z1, Z2 ... The value of the argument must be 'complex' arithmetic

For example, the definition of the 'trunc' function requires that it have the form:

trunc(R)

Therefore, the use of an argument that does not yield a 'real' arithmetic value is invalid.

Each definition gives the precision attribute of the result without regard for the capacity of a particular implementation of PL/I. For each implementation of PL/I, the defined precision must be limited to the capacity of the host computer. The defined precision is modified to a Multics precision as follows:

<u>Defined</u>	<u>Multics</u>
fixed bin(p,q)	fixed bin(min(max(p,1),71),min(max(q,-128),127))
float bin(p)	float bin(min(max(p,1),63))
fixed dec(p,q)	fixed dec(min(max(p,1),59),min(max(q,-128),127))
float dec(p)	float dec(min(max(p,1),59))

These rules simply require that the precision of the result of an operation lie within the ranges for the number-of-digits and scale-factor that were given earlier, in Section III, "Value Storage."

COMMON DATA ATTRIBUTES

Many of the definitions that follow refer to the common data attributes of the arguments. These attributes are as follows:

- The mode attribute is 'complex' if the modes of the arguments differ; otherwise, it is the common mode of all the arguments.
- The scale attribute is 'float' if the scales of the arguments differ; otherwise, it is the common scale of all the arguments.
- The base attribute is 'binary' if the bases of the arguments differ; otherwise, it is the common base of all the arguments.

For example, suppose that an operation has two arguments with the following data types:

real fixed dec(10)

real float bin(27)

Then the common data attributes are:

real float bin

Elementary Operations

There are five operators and four functions that are used to perform the elementary operations of arithmetic. They are:

- The plus and minus operators. Each can be used either as a prefix operator to supply the sign of an expression or as an infix operator to add or subtract one expression from another.
- The multiplication, division, and exponentiation operators. The exponentiation operator accepts noninteger exponents.
- The 'add', 'subtract', 'multiply', and 'divide' functions, which are used in fixed-point calculations when it is necessary to specify the precision of the result.

These functions are of general interest.

PREFIX SIGN OPERATORS

Parenthesized operator expressions for the prefix sign operators have the forms:

$$\begin{aligned} & (+ X) \\ & (- X) \end{aligned}$$

The result is the unchanged value or the negated value, respectively, of X. The storage type of the result is the same as that of X. Examples are:

$$\begin{aligned} + (+0435.241) &= +0435.241 \\ - (+0435.241) &= -0435.241 \\ + (-2.8192e0) &= -2.8192e0 \\ - (-2.8192e0) &= +2.8192e0 \\ + (+1.3000e0-17.891e0i) &= +1.3000e0-17.891e0i \\ - (+1.3000e0-17.891e0i) &= -1.3000e0+17.891e0i \end{aligned}$$

INFIX SIGN OPERATORS

Parenthesized operator expressions for the infix sign operators have the forms:

$$\begin{aligned} & (X1 + X2) \\ & (X1 - X2) \end{aligned}$$

The result is the sum or difference, respectively, of the operands. Before the operation is performed, the operands are converted to the common data attributes, defined earlier. The result has the same data type as the converted operands except:

- If the converted operands are 'float' with precisions '(p1)' and '(p2)', then the result has the precision:
$$(\max(p1,p2))$$
- If the converted operands are 'fixed' with precisions '(p1,q1)' and '(p2,q2)', then the result has the precision:
$$(\max(p1-q1,p2-q2)+\max(q1,q2)+1, \max(q1,q2))$$

Observe that a 'fixed' result has a digit position for each digit position in either operand and, further, an extra, high-order digit position.

Examples are:

```
(2.000e0) + (2.0e0)      = 4.000e0
(2.000e0) + (2)          = 4.000e0
(2.000e0) + (00010.0b)  = 100.00000000000e0b
(2.000e0) + (2+0i)      = 4.000e0+0.000e0i

(5555) + (.333)         = 05555.333
(5555) + (.00023)      = 05555.00023
```

In the last example, each slashed zero is a filler zero; therefore, the second argument has precision '(2,5)', not '(5,5)'.

MULTIPLICATION OPERATOR

A parenthesized operator expression for the multiplication operator has the form:

```
( X1 * X2 )
```

The result is the product of the operands. Before the operation is performed, the operands are converted to the common data attributes defined earlier. The result has the same data type as the converted operands except:

- If the converted operands are 'float' with precisions '(p1)' and '(p2)', then the result has the precision:

```
( max(p1,p2) )
```

- If the converted operands are 'fixed' with precisions '(p1,q1)' and '(p2,q2)', then the result has the precision:

```
( p1+p2+1, q1+q2 )
```

Observe that a 'fixed' result has more digits than either operand; thus multiplications tend to increase the precision of an expression.

Examples are:

```
(2.000e0) * (-3.0e0) = -6.000e0
(002.00) * (-003.00) = -0000006.0000
(002.00) * (-003.00) * (004.00) = -00000000024.000000
```

DIVISION OPERATOR

A parenthesized operator expression for the division operator has the form:

$$(X1 / X2)$$

The result is the quotient of X1 by X2. If X2 is zero, then the 'zerodivide' condition occurs, and any attempt to resume execution at the division operation is invalid. Before the operation is performed, the operands are converted to the common data attributes defined earlier. The result has the same data type as the converted operand except:

- If the converted operands are 'float' with precision '(p1)' and '(p2)', then the result has the precision:

$$(\max(p1,p2))$$

- If the converted operands are 'fixed' with precisions '(p1,q1)' and '(p2,q2)', respectively, then the result has the precision:

$$(N, N-p1+q1-q2)$$

where N is the maximum number-of-digits, 59 for 'decimal' and 71 for 'binary'. Observe that a 'fixed' result always is of maximum size.

The precision of a 'fixed' result is of maximum size because PL/I cannot make a better choice on the basis of available information. In virtually all cases, the programmer will have better information; for fixed-point division, he should use the 'divide' function, which allows specification of the correct result precision. Examples of the division operator are:

$$(6.000e0) / (-3.0e0) = -2.000e0$$
$$1/3 = 0.333...3 \quad (59 \text{ digits})$$
$$2+1/3 = 02.333...3 \quad (60 \text{ digits}) = 2.333...3 \quad (59 \text{ digits})$$
$$22+1/3 = 22.333...3 \quad (60 \text{ digits}) = (\text{size})$$

The fact that the small and useful expression '22+1/3' cannot be used without producing an occurrence of the 'size' condition shows the weakness of the fixed-point divide operation. The expression could be written as:

$$22+\text{divide}(1,3,10,10) = 022.3333333333$$

EXPONENTIATION OPERATOR

A parenthesized operator expression for the exponentiation operator has the form:

$X1 ** X2$

The result is $X1$ to the power $X2$. For some values, the mathematical value of the operator is indeterminate; for these values the PL/I operator is defined as follows:

- When $X1$ is zero, the function is defined as follows:
if $X2$ is 'real' and $X2 > 0$ or
 $X2$ is 'complex' and $\text{real}(X2) > 0$ and $\text{imag}(X2) = 0$
then $0 ** X2 = 0$;
otherwise, the 'error' condition occurs.
- When $X2$ is zero and $X1$ is not zero, the function is defined as follows:
 $X1 ** 0 = 1$
- When $X1$ is 'real' and negative, the function is defined as follows:
if $X2$ is a decimal integer constant (without a sign),
then $(-X1) ** X2 = ((-1) ** X2) * (X1 ** X2)$;
otherwise, the 'error' condition occurs.

This restriction excludes a complex value for a 'real' result; but observe that if $X2$ is supplied in any way other than a constant, the 'error' condition occurs (even if its value is a positive integer).

Before the operation is performed, the operands are converted to the common data attributes defined earlier; however, for exponentiation the following exception applies:

If the second argument, the exponent, has the original data type:

real fixed base(p2,0)

then the adjusted data type for the second operand retains the attributes 'real' and 'fixed' regardless of the attributes of the first operand.

This exception is perceptible only when a conversion error can occur; otherwise, it can be ignored.

'add', 'subtract', 'multiply', AND 'divide' FUNCTIONS

References to these functions have the forms:

add(X1,X2,P)	add(X1,X2,P,Q)
subtract(X1,X2,P)	subtract(X1,X2,P,Q)
multiply(X1,X2,P)	multiply(X1,X2,P,Q)
divide(X1,X2,P)	divide(X1,X2,P,Q)

where P and Q cannot be general expressions but must instead be given as decimal integer constants (Q can be signed). These functions are equivalent to the corresponding operators in all respects but one:

The precision of the result of each of these functions is given explicitly by P (number of digits) and Q (scaling factor).

If the result is 'fixed' and Q is not given, then Q=0 is assumed. If the result is 'float' and Q is given, then the reference is invalid.

Comparison Operations

There are eight operators and two functions that are used to compare arithmetic values. They are:

- The relational operators, including '=', '<', and six similar operators. Only the application of these operators to arithmetic values is described here.
- The 'min' and 'max' functions, which yield the smallest or largest member, respectively, of a given list of arithmetic values.

These functions are of general interest.

RELATIONAL OPERATORS FOR ARITHMETIC VALUES

A parenthesized expression for a relational operator has the form:

(X1 op X2)

where the operator, op, is one of the following:

=	(is equal to)
<	(is less than)
>	(is greater than)
<=	(is less than or equal to)
>=	(is greater than or equal to)
^=	(is not equal to)
^<	(is not less than)
^>	(is not greater than)

The arguments must be 'real' unless the operator is '=' or '^='. The relational operators can apply to many types of values, as follows:

- If either operand yields an arithmetic value or a character string declared with a numeric picture, then the expression is an arithmetic comparison and is described here.
- If both operands yield string values and neither is declared with a numeric picture, then the expression is a string comparison and is described later in this section under "The Relational Operators for String Values".
- If both operands yield address values, then the expression is an address comparison and is described later in this section under "The Relational Operators for Address Values".

The result of a comparison expression is "1"b or "0"b depending on whether the comparison is true or false. The arguments are converted to the common data attributes. The result has storage type 'bit(1) nonvarying'. Examples are:

```
+015.03 = +15.0300 = "1"b
+015.03 < +15.0300 = "0"b
+015.03 <= +15.0300 = "1"b

+15.030e0 = +15.03 = "1"b
+15.030e0 < +15.031 = "1"b
+15.030e0 ^< +15.031 = "0"b
```

'min' AND 'max' FUNCTIONS

References to these functions have the forms:

```
min(R1,R2,...)
max(R1,R2,...)
```

where the arguments must be 'real' and where each function can have any number of arguments, provided it has more than one argument. The result is the minimum value or the maximum value, respectively, in the set of values given by the arguments. The arguments must yield 'real' values; these values are converted to the common data attributes defined earlier in this section. The result has the storage type of the converted arguments except:

- If the converted arguments, R1, R2, and so on, are 'float' and have precisions '(p1)', '(p2)', and so on, then the precision of the result is:

```
( max(p1,p2,...) )
```

- If the converted arguments, R1, R2, and so on, are 'fixed' and have precisions '(p1,q1)', '(p2,q2)', and so on, the precision of the result is:

```
( max(pi-q1,p2-q2,...), max(q1,q2,...) )
```

These expressions provide the precision that is necessary to accommodate any argument without loss of digits. Examples are:

```
min(2.000e0,-000.5,11.1000000e0) = -5.00000000e-1
max(-3,99,99.05,00.013) = 99.050
```

Truncating Functions

There are five functions that change a value by discarding digits from a representation of the value. They are:

- The 'trunc', 'floor', and 'ceil' functions, which use three different rules for discarding all fractional digits.
- The 'round' function, which discards any sequence of low order digits by rounding.
- The 'mod' function, which, in one form of usage, discards any sequence of high order digits.

These functions are of general interest.

'trunc', 'floor', AND 'ceil' FUNCTIONS

References to these functions have the forms:

```
trunc(R)
floor(R)
ceil(R)
```

where the argument must be 'real'. The results of these functions are the truncation, the floor, and the ceiling, respectively, of R. Each function discards the fractional part of its argument to produce an integer-valued 'real' result.

- If R is an integer:
trunc(R) = R
floor(R) = R
ceil(R) = R
- If R is not an integer:
trunc(R) is the result of discarding the fractional digits of R
floor(R) is the next integer below R
ceil(R) is the next integer above R

The result has the storage type of the argument except that, for a 'fixed' argument, the precision '(p,q)' becomes:

$(\max(p-q+1,1),0)$

That is, a fixed result is an integer with an added high-order digit. Examples of the functions are:

<u>R</u>	<u>trunc(R)</u>	<u>floor(R)</u>	<u>ceil(R)</u>
1.782e0	1.000e0	1.000e0	2.000e0
001	0001	0001	0001
-9.56	-09.	-10.	-09.
-.0003	+0.	-1.	+0.

Observe that 'floor(-9.56)' makes use of the added high-order digit of the result.

'round' FUNCTION

A reference to the function has the form:

round(X,Q)

where Q cannot be a general expression but must instead be given as an optionally signed decimal integer. The result has the same mode as X and is a rounding of the value of X. Rounding is defined as follows:

- When a value is rounded to n digits, the digits after the nth digit are dropped and the nth digit is increased by one if the (n+1)th digit is '5' or greater (for decimal) or '1' (for binary).

When X is 'real', the result of the function is defined as follows:

- If X is 'float', Q must be greater than 0 and the mantissa is rounded to Q digits.
If X is 'fixed', it is rounded to a value that has Q fractional digits.

The result has the same storage type as X except that:

- If X is 'float', the precision of the result is '(Q)'.
If X is 'fixed' with precision '(p,q)', the precision of the result is
 $(p-q+1+Q,Q)$

Thus a high-order digit is added in case the rounding propagates to a new order of magnitude.

Examples for floating-point are:

```
round(-183.629e6,4)    = -183.6e6
round(-183.629e6,5)    = -183.63e6
round(-183.629e6,6)    = -183.629e6
round(-111010.111e3b,7) = -111011.0e3b
```

Observe that neither the exponent of the value nor the point in the mantissa affect the rounding.

Examples for fixed point are:

```
round(-183.629,2)     = -0183.63
round(-183.629,-1)    = -0180. (fixed dec(3,-1))
round(-183.629,-5)    = +000000. (fixed dec(1,-5))
```

In these examples, a slashed zero is a filler zero. In the last example, the formula for the precision gives:

$$(p-q+1+Q,Q) = (6-3+1-5,-5) = (-1,-5)$$

However, the number-of-digits must be greater than zero, so the precision is '(1,-5)'. Observe that the position of the point does affect the rounding of fixed values.

For 'complex' values, the function is defined by:

$$\text{round}(R1+R2*1i,Q) = \text{round}(R1,Q) + \text{round}(R2,Q)*1i$$

For example:

$$\text{round}(21.56+06.21i,0) = 022.+006.i$$

'mod' FUNCTION

A reference to the function has the form:

$$\text{mod}(R1,R2)$$

where the arguments must be 'real'. The result is 'real' and is $R1 \bmod R2$, that is:

- If $R2 \neq 0$, then $\text{mod}(R1,R2) = R1 - R2*\text{floor}(R1/R2)$
- If $R2 = 0$, then $\text{mod}(R1,R2) = R1$

The arguments must yield 'real' values; these values are converted to the common data attributes, defined earlier. The result has the storage type of the converted arguments, except:

- If the converted arguments are 'float' with precisions '(p1)' and '(p2)', respectively, then the result has the precision:
(max(p1,p2))
- If the converted arguments are fixed and have precisions '(p1,q1)' and '(p2,q2)', respectively, then the result has precision:
(p2-q2+max(q1,q2),max(q1,q2))

Thus the result has as many integer digits as R2 and enough fractional digits for either R1 or R2.

When both R1 and R2 are positive, the modulus is the remainder of the conventional integer division of R1 by R2. For example:

```
mod(42,5) = 2
mod(129.2867,25.0) = 04.2867
mod(129.2867e0,25.0e0) = 004.2867e0
mod(182.00e0,3) = 002.00e0
```

The storage types for these examples are:

<u>Converted Arguments</u>	<u>Result</u>
fixed dec(2,0), fixed dec(1,0)	fixed dec(1,0)
fixed dec(7,4), fixed dec(3,1)	fixed dec(6,4)
float dec(7), float dec(3)	float dec(7)
float dec(5), float dec(5)	float dec(5)

When R1 or R2 is negative, the notion of a remainder is not well-defined, so the behavior of the function is less obvious. Consider the four possible combinations of signs:

```
mod(42,5) = 2 (floor(42/5 = 8))
mod(42,-5) = -3 (floor(42/-5 = -9))
mod(-42,5) = 3 (floor(-42/5 = -9))
mod(-42,-5) = -2 (floor(-42/-5 = 8))
```

When R1 is positive and R2 = 10**n, the function discards digits from the left end of a 'decimal' value until R1<R2. For example:

```
mod(231.34,100) = 031.34
mod(231.34,1) = 0.34
mod(231.34,.1) = .04
```

Similarly, for 'binary' values:

```
mod(111110.110b,1000b) = 0110.110b
mod(111110.110b,.1b)  = .010b
```

Observe that the use of the 'mod' function for truncation does not work when R1 is negative. That is,

```
mod(-231.34,1) = 0.66
```

Sign-Manipulation Functions

There are two functions that manipulate the sign of an arithmetic variable. They are:

- The 'abs' function, which sets the sign of a real number to plus and forms the modulus of a complex number.
- The 'sign' function, which yields a value that is +1 or -1 depending on the sign of the argument.

These functions are of general interest.

'abs' FUNCTION

A reference to the function has the form:

```
abs(X)
```

The result is the absolute value of X. For a 'real' argument, the result has the same storage type as the converted argument. Examples are:

```
abs(-13.284)   = +13.284
abs(+0019.8)  = +0019.8
abs(-3.0000e2) = +3.0000e2
abs(-63.000)  = +63.000
```

For 'complex' values, the function is defined by:

```
abs(R1+R2*1i) = + sqrt(R1**2 + R2**2)
```

The result has the same data type as the 'complex' argument except that the mode is 'real' and, for a fixed argument, the precision '(p,q)' becomes '(p+1,q)'.

Consider the function reference:

```
abs(+8.00+9.00i) = +12.04
```

Observe that the result value in this example makes use of the added digit in the result. Other examples are:

```
abs(-003.00+003.00i) = +0004.24
abs(+20.200e0+.80000e0i) = +20.216e0
abs(-63.000+00.000i) = +063.000
```

Compare the last example to the last example for 'real' arguments: the values are the same but the number-of-digits is different.

'sign' FUNCTION

A reference to the function has the form:

```
sign(R)
```

where the argument must be 'real'. The result is +1, +0, or -1 depending on whether R is positive, zero, or negative. The result is a 17-bit integer. Examples are:

```
sign(-019.32) = -1. (as a 'real fixed bin(17)' value)
sign(+8.23e2) = +1. (as a 'real fixed bin(17)' value)
sign(+0000.0) = +0. (as a 'real fixed bin(17)' value)
```

Observe that, if R is 'float':

```
R = sign(R)*abs(R)
```

However, if R is not 'float' the two sides of this equation agree in value but not in data type.

Complex Arithmetic Functions

There are three functions that are used to manipulate the real and imaginary parts of a 'complex' value. They are:

- The 'real' and 'imag' functions, which yield the real and imaginary parts of a complex value as real numbers. These functions can also be used as pseudo-variables, as described in Section X, "Value Assignment."
- The 'complex' function, which yields a complex value from given real parts.
- The 'conjg' function, which yields the conjugate of a given complex value.

These functions are used only in computations in complex arithmetic.

'real' AND 'imag' FUNCTIONS FOR ARITHMETIC

References to these functions have the form:

```
real(Z)
imag(Z)
```

where the argument must be 'complex'. The result is the real part or imaginary part, respectively, of Z. The result has the same storage type as the argument except the result is 'real'. For example:

```
real(-23041.e-2+00519.e1i) = -23041.e-2
imag(-23041.e-2+00519.e1i) = +00519.e1
```

The use of these functions for conversion is described later in this section, under "Conversion Operations".

The pseudo-variables named 'real' and 'imag' are described later, in Section X, on "Value Assignment."

'complex' FUNCTION

A reference to this function has the form:

```
complex(R1,R2)
```

where the arguments must be 'real'. The result is a complex value whose real part is the value of R1 and whose imaginary part is the value of R2. Before interpretation of the function, R1 and R2 are converted to arithmetic values of the common data attributes defined earlier. The result has the same storage type as the converted arguments except that the result is 'complex'. For example:

```
complex(-23041.e-2,+00519e1) = -23041.e-2+00519.e1i
complex(-23041e-2,0)         = -23041.e-2+00000.e0i

complex(0,+00519.e1) = +00000.e0+00519.e1i

complex(16,"10111b")      = 0...010000b+0...01011bi (71 digits each)
complex(16,"23041.e-2")  = 0...016+0...230i (59 digits each)
```

'conjg' FUNCTION

A reference to the function has the form:

conjg(Z)

where the argument must be 'complex'. The result is the conjugate of Z, that is:

conjg(R1+R2*1i) = R1-R2*1i

The result has the same storage type as the argument. For example:

```
conjg(-05.000+02.000i) = -05.000-02.000i
conjg(+2.312e0-.4231e0i) = +2.312e0+.4231e0i
conjg(+0002+0000i) = +0002+0000i
```

MATHEMATICAL OPERATIONS

The functions described here always produce floating-point results. These functions contrast with the arithmetic functions, which produce fixed-point results for some arguments and floating-point results for others.

These functions are called "mathematical" because they are usually required only in mathematical, scientific, and engineering applications of PL/I. They are all transcendental functions and cannot, therefore, be defined in terms of a single expression using the basic arithmetic operations.

Conventions for Definitions

The functions defined here have many points in common. The data types of arguments and results are handled in a uniform way, and the same methods are applied to the calculation of the result value. Therefore, the following general conventions can be applied to all of the definitions.

ARGUMENTS

All of the functions operate on 'float' arguments. If an argument is 'fixed', then its value is converted to a 'float' target whose number-of-digits is the same as that of the given argument. For example:

<u>Argument</u>	<u>Target for Conversion</u>
real fixed bin(30,5)	real float bin(30)
complex fixed dec(10,0)	complex float dec(10)
complex fixed bin (70,0)	complex float bin(63)

The last example shows that the number-of-digits is not allowed to exceed the maximums for 'float' values (63 for 'binary' and 59 for 'decimal').

For a two-argument function ('atan' and 'atand'), the arguments are converted to a 'float' target with a common base. If the arguments have different bases, then the target base is 'binary'.

Some of the mathematical built-in functions require 'real' arguments. The following symbols are used in the definitions:

R, R1, R2, ... The mode of the operand must be 'real'.

X, X1, X2, ... The mode of the operand is not restricted.

If a 'complex' argument is used where a 'real' argument is required, the function reference is invalid. An implicit conversion from 'complex' to 'real' is not assumed.

RESULTS

The storage type of the result of a mathematical function is the same as the storage type of its converted argument(s).

EVALUATION

The Multics implementation of the mathematical functions uses binary arithmetic. If the argument is 'decimal(p)', then it is converted to 'binary(63)'. When the result of the function is obtained, it is converted back to 'decimal(p)'. As a result of this policy, a decimal result cannot be accurate to more than about 20 decimal digits.

Some functions are not defined for certain arguments. In such cases, the definition of the function says that the 'error' condition occurs. The 'error' condition is used for any error that does not have its own, more specialized, condition. When the 'error' condition occurs, the execution of the program cannot resume at the point of interruption, so some disruption is inevitable. The details are given in Section XIII, "Condition Handling."

A function can have a 'complex' result if and only if its argument is 'complex'. Thus a 'real' argument that would produce a 'complex' result is an error. Consider 'sqrt(X)'. If X is 'real' and $X < 0$, then the 'error' condition occurs; but if X is 'complex', the 'error' condition does not occur.

Some functions are potentially multi-valued. For such a function, there is more than one result value that satisfies the mathematical definition of the function. In each such case, PL/I imposes conditions on the result that provide a unique value. For example, in the definition of 'sqrt', the following condition is given:

For X 'real': $\text{sqrt}(X) \geq 0$

Thus the negative square root is excluded and the result is uniquely defined.

The mathematical definitions of the functions as they apply to real arguments are not given; it is assumed that the reader who uses these functions is familiar with their definition. On the other hand, the definitions of the functions as they apply to complex arguments are given. These definitions are given because, although they are part of standard mathematics, they are not easy to find. The definitions are expressed in terms of operations on 'real' values and make use of valid PL/I expressions.

In some of the definitions, the mathematical constants 3.14159... and 2.71828... are referred to by the names pi and e, respectively. These names are not built into the PL/I language. If a programmer needs these names, he must declare and set them just as he would any other variable.

Functions Related to Exponentiation

There are five built-in functions that relate to the exponentiation operation. They are:

- The 'exp' function, which raises the mathematical constant e = 2.71828... to a given power.
- The 'log' function, which is the inverse of the 'exp' function.
- The 'log10' and 'log2' functions, which are the inverses of 10**R and 2**R, respectively.
- The 'sqrt' function, which raises a given number to the 1/2 power.

These functions are fundamental to scientific and engineering applications.

'exp' FUNCTION

A reference to the function has the form:

exp(X)

The result is e raised to the power X, where e is the base of the system of natural logarithms. Examples are:

```
exp(1.0000e0) = 2.7183e0
exp(0.0000e0) = 1.0000e0
exp(-1.0000e0) = .36788e0
```

For complex values, the function is defined by:

$$\text{exp}(R1+R2*1i) = \text{exp}(R1)*(\cos(R2)+\sin(R2)*1i)$$

Thus, for example:

$$\text{exp}(0.0000e0+.78540e0i) = .70711e0+.70711e0i$$

'log' FUNCTION

A reference to the function has the form:

$\log(X)$

The result is the logarithm base-e, or natural logarithm, of X. The function is the inverse of the 'exp' function. Examples are:

```
log(2.7183e0) = 1.0000e0
log(1.0000e0) = 0.0000e0
log(.36788e0) = -1.0000e0
```

The argument is restricted as follows:

- If X is 'real', then $X < 0$ causes the 'error' condition to occur because the result is a complex value.
- If X is 'real' or 'complex', $X = 0$ causes the 'error' condition to occur because the result is not defined.

The result satisfies the conditions:

For X 'real': The requirement that the result must be 'real' is sufficient to make the result unique.

For X 'complex': $-\pi < \text{imag}(\log(X)) \leq \pi$

For complex values, the function is defined by:

```
log(R1+R2*1i) = .5*log(R1**2+R2**2) + atan(R2,R1)*1i
```

Thus, for example:

```
log(.70711e0+.70711e0i) = .45481e-5+.78540e0i
```

(observe that .70711 is approximately $1/\sqrt{2}$ and .45481e-5 is close to zero.)

'log10' AND 'log2' FUNCTIONS

References to these functions have the forms:

```
log10(R)
log2(R)
```

where the argument must be 'real'. The results are the logarithm base-10 and logarithm base-2, respectively, of R. Examples are:

```
log10(1.0000e4) = 4.0000e0
log10(2.0000e4) = 4.3010e0
log2(64.0000e0) = 6.0000e0
```

The following restriction applies to the argument:

- For both 'log10' and 'log2', if $R \leq 0$, then the error condition occurs because a real result does not exist.

These functions can be used to determine how many digits are required for the representation of a given integer in either decimal or binary representation. For example:

$$\text{ceil}(\log_2(70)) = \text{ceil}(6.1) = 7$$

which shows that 70 requires seven digits for its binary representation.

'sqrt' FUNCTION

A reference to the function has the form:

$$\text{sqrt}(X)$$

The result is a square root of X. Examples are:

$$\begin{aligned}\text{sqrt}(0.0000e0) &= 0.0000e0 \\ \text{sqrt}(1.0000e0) &= 1.0000e0 \\ \text{sqrt}(2.0000e0) &= 1.4142e0\end{aligned}$$

The argument is restricted as follows:

If X is 'real', then $X < 0$ causes the 'error' condition to occur because the result is a complex value.

The result satisfies the conditions:

$$\begin{aligned}\text{For } X \text{ 'real':} & \quad \text{sqrt}(X) \geq 0 \\ \text{For } X \text{ 'complex':} & \quad \text{either } \text{real}(\text{sqrt}(X)) > 0 \\ & \quad \text{or } \text{real}(\text{sqrt}(X)) = 0 \text{ and } \text{imag}(\text{sqrt}(X)) \geq 0\end{aligned}$$

For complex values, the function is defined by:

$$\text{sqrt}(R_1 + R_2 * 1i) = \text{sqrt}((R_3 + R_1)/2) + \text{sqrt}((R_3 - R_1)/2) * 1i$$

where:

$$R_3 = \text{sqrt}(R_1^2 + R_2^2)$$

Thus, for example:

$$\text{sqrt}(3.0000e0 + 4.0000e0i) = 2.0000e0 + 1.0000e0i$$

Trigonometric Functions

There are 12 built-in functions for trigonometry. They are:

- The 'sin', 'cos', and 'tan' functions, which assume that their arguments are in radians.
- The 'sind', 'cosd', and 'tand' functions, which assume that their arguments are in degrees.
- The ordinary 'atan' and 'atand' functions, which each have one argument, and are the conventional inverses of the 'tan' and 'tand' functions.
- The ordinary 'acos' and 'asin' functions, which assume that their arguments are in radians and are the conventional inverses of 'cos' and 'sin' functions.
- The Cartesian 'atan' and 'atand' functions, which each have two arguments and have results that range over all four quadrants of the coordinate system.

The functions are not the usual textbook selection; instead, the functions are selected and designed for the needs of practical computing. For example, the secant function is omitted because it is rarely used and can be expressed in terms of the 'cos' built-in function. On the other hand, the Cartesian 'atan' function, rarely mentioned in trigonometry texts, is included because it is needed for calculation of complex values and certain other calculations.

'sin', 'cos', AND 'tan' FUNCTION

References to these functions have the forms:

```
sin(X)
cos(X)
tan(X)
```

The result is the sine, cosine, of tangent, respectively, of X. X is assumed to be expressed in radians. Examples are:

```
sin(0.0000e0) = 0.0000e0
cos(3.1416e0) = -1.0000e0
tan(.78540e0) = 1.0000e0
```

For complex values, the function is defined by:

```
sin(R1+R2*1i) = sin(R1)*cosh(R2) + cos(R1)*sinh(R2)*1i
cos(R1+R2*1i) = cos(R1)*cosh(R2) - sin(R1)*sinh(R2)*1i
tan(R1+R2*1i) = sin(R1+R2*1i)/cos(R1+R2*1i)
```

Thus, for example:

```
sin(0.0000e0+1.0000e0i) = 0.0000e0+1.1752e0i
cos(0.0000e0+1.0000e0i) = 1.5431e0+0.0000e0i
tan(0.0000e0+1.0000e0i) = 0.0000e0+.76159e0i
```

ORDINARY 'acos' FUNCTION

A reference to the function has the form:

$\text{acos}(x)$

It takes a single, real argument and the result is the arccosine, expressed in radians, of x . Examples are:

$\text{acos}(1.0000000e0) = 0.0000000e+0$
 $\text{acos}(0.0000000e0) = 1.5707963e+0$
 $\text{acos}(-1.0000000e0) = 3.1415926e+0$

The argument is restricted as follows:

x must be ≤ 1 and ≥ -1

The result satisfies the conditions:

$0 \leq \text{acos}(x) \leq \pi$

ORDINARY 'asin' FUNCTION

A reference to the function has the form:

$\text{asin}(x)$

It takes a single, real argument and the result is the arcsine expressed in radians of x . Examples are:

$\text{asin}(1.0000000e0) = 1.5707963e+0$
 $\text{asin}(0.0000000e0) = 0.0000000e+0$
 $\text{asin}(-1.0000000e0) = 1.5707963e+0$

The argument is restricted as follows:

x must be ≤ 1 and ≥ -1

The result satisfies the conditions:

$-\pi/2 \leq \text{asin}(x) \leq \pi/2$

This page intentionally left blank.

'sind', 'cosd', AND 'tand' FUNCTIONS

References to these functions have the forms:

```
sind(R)
cosd(R)
tand(R)
```

where the argument must be 'real'. The result is the sine, cosine, or tangent, respectively, of R. R is assumed to be expressed in degrees. For example:

```
sind(0.0000e0) = 0.0000e0
cosd(180.00e0) = -1.0000e0
tand(45.000e0) = 1.0000e0
```

ORDINARY 'atan' FUNCTION

A reference to the function has the form:

```
atan(X)
```

The result is the arctangent, expressed in radians, of X. Examples are:

```
atan(1.0000e0) = .78540e0
atan(0.0000e0) = 0.0000e0
atan(1.0000e10) = 1.5708e0
```

The argument is restricted as follows:

If X is 'complex', then X=+1i or X=-1i causes the 'error' condition to occur because the result is not defined.

The result satisfies the conditions:

```
For X 'real':      -pi/2 < atan(X) < pi/2
For X 'complex':  -pi/2 < real(atan(X)) < pi/2
```

For complex values, the function is defined in terms of the 'atanh' built-in function:

```
atan(X) = -1i*atanh(X*1i)
```

Thus, for example:

```
atan(0.0000e0-2.0000e0i) = 1.5708e0-.54931e0i
```

where this result is approximately:

```
pi/2 - (1/2)*log(3)*1i
```

ORDINARY 'atand' FUNCTION

A reference to the function has the form:

atand(R)

where the argument must be 'real'. The result is the arctangent, expressed in degrees, of X. Examples are:

```
atand(1.0000e0) = 45.000e0
atand(0.0000e0) = 0.0000e0
atand(1.0000e10) = 90.000e0
```

The result satisfies the condition:

$-90 < \text{atand}(R) < 90$

CARTESIAN 'atan' FUNCTION

A reference to the function has the form:

atan(R1,R2)

The arguments must be 'real'. This function can be used to calculate the angular coordinate, in radians, of a point in the X-Y plane. Specifically,

Suppose a point is given with the coordinates:

```
y = R1
x = R2
```

Then the result of the function reference 'atan(R1,R2)' is the angle of a line that begins at the origin of the coordinate system and passes through the given point. The angle is expressed in radians, is measured counter-clockwise from the X axis, and is in the range:

$-\pi < \text{atan}(R1,R2) \leq \pi$

Because this function puts the angle in the correct quadrant, it simplifies calculations. Observe, however, that the arguments are written in y,x order; this is contrary to conventional mathematical practice, which usually gives coordinates in x,y order. Some examples of the function are:

```
atan(0.0000e0,1.0000e0) = 0.0000e0
atan(1.0000e0,1.0000e0) = .78540e0 (= (1/4)*pi)
atan(9.0000e2,9.0000e2) = .78540e0 (= (1/4)*pi)
atan(1.0000e0,0.0000e0) = 1.5708e0 (= (1/2)*pi)
atan(1.0000e0,-1.0000e0) = 1.3562e0 (= (3/4)*pi)
atan(0.0000e0,-1.0000e0) = 3.1416e0 (= pi)
atan(-1.0000e0,-1.0000e0) = -2.3562e0 (= -(3/4)*pi)
```

The arguments must be 'real'; furthermore:

R1 = 0 and R2 = 0 cause the 'error' condition to occur because the function is not defined at the origin of the coordinate system.

The function can be defined in terms of the ordinary 'atan' built-in function as follows:

	atan(y/x)	for any y and x > 0
	pi/2	for y > 0 and x = 0
atan(y,x) =	atan(y/x)+pi	for y >= 0 and x < 0
	atan(y/x)-pi	for y < 0 and x < 0
	-pi/2	for y < 0 and x = 0

The function can be used to obtain the argument of a complex value; that is:

Arg(X) = atan(imag(X),real(X))

Observe, again, that the order of arguments is the reverse of the usual order.

CARTESIAN 'atand' FUNCTION

A reference to the function has the form:

atand(R1,R2)

where the arguments must be 'real'. The function is used to compute the angular coordinate, in degrees, of a point in the X-Y plane. The function is defined in terms of the Cartesian 'atan' function, as follows:

atand(R1,R2) = (180/pi)*atan(R1,R2)

Examples are:

atand(0.0000e0,1.0000e0)	=	0.0000e0
atand(1.0000e0,1.0000e0)	=	45.000e0
atand(1.0000e0,-1.0000e0)	=	135.00e0
atand(0.0000e0,-1.0000e0)	=	180.00e0
atand(-1.0000e0,-1.0000e0)	=	135.00e0

The following restriction applies to the argument:

R1 = 0 and R2 = 0 causes the 'error' condition to occur.

Hyperbolic Functions

There are four built-in hyperbolic functions. They are:

- The 'sinh', 'cosh', and 'tanh' functions
- The 'atanh' function, which is the inverse of the 'tanh' function

These functions are used primarily for calculations with complex numbers.

'sinh', 'cosh', AND 'tanh' FUNCTIONS

References to these functions have the forms:

```
sinh(X)
cosh(X)
tanh(X)
```

The result is the hyperbolic sine, hyperbolic cosine, or hyperbolic tangent, respectively, of X. It is assumed that X is expressed in radians. Examples are:

```
sinh(1.0000e0) = 1.1752e0
cosh(1.0000e0) = 1.5431e0
tanh(1.0000e0) = .76159e0
```

For complex values, the function is defined by:

```
sinh(R1+R2*1i) = sinh(R1)*cos(R2) + cosh(R1)*sin(R2)*1i
cosh(R1+R2*1i) = cosh(R1)*cos(R2) + sinh(R1)*sin(R2)*1i
tanh(R1+R2*1i) = sinh(R1+R2*1i)/cosh(R1+R2*1i)
```

Thus, for example:

```
sinh(0.0000e0+.78540e0i) = 0.0000e0+.70711e0i
cosh(0.0000e0+.78540e0i) = .70711e0+0.0000e0i
tanh(0.0000e0+.78540e0i) = 0.0000e0+1.0000e0i
```

(where .78540 is an approximation of pi/4.)

'atanh' FUNCTION

A reference to this function has the form:

atanh(X)

The result is the arc hyperbolic tangent, expressed in radians, of X. Examples are:

atanh(0.0000e0) = 0.0000e0
atanh(.76159e0) = 1.0000e0

The argument is restricted as follows:

If X is 'real', then $\text{abs}(X) \geq 1$ causes the 'error' condition to occur because the result is a complex value.

If X is 'complex', then $X = +1$ or $X = -1$ causes the 'error' condition to occur because the result is not defined.

The result satisfies the conditions:

For X 'real': (The requirement that the result must be 'real' is sufficient to make the result unique.)

For X 'complex': $-\pi/2 < \text{imag}(\text{atanh}(X)) \leq \pi/2$

For complex values, the function is defined in terms of the 'log' built-in function:

$\text{atanh}(X) = \log((1+X)/(1-X))/2$

Functions for Statistical Analysis

There are two built-in functions that relate to statistical calculations. They are:

- The 'erf' function, which is the error function
- The 'erfc' function, which is the complement of the error function.

'erf' AND 'erfc' FUNCTIONS

References to these functions have the form:

```
erf(R)
erfc(R)
```

where the argument must be 'real'. The results are the error function and the error function complement, respectively, of X. For example:

```
erf(0.000e0) = 0.000e0
erf(1.000e0) = .8427e0
erfc(0.000e0) = 1.000e0
```

The functions are defined by the equations:

$$\text{erf}(R) = (2/\sqrt{\pi}) * \int_0^R e^{-(t^2)} dt$$

$$\text{erfc}(R) = 1 - \text{erf}(R)$$

STRING OPERATIONS

The string operations manipulate string-related values. A string-related value is:

```
a string,
an integer that gives a position in a string, or
an integer that is the length of a string.
```

Each string operation converts its operands or arguments to string-related values and then computes a result that is also a string-related value.

Conventions for Definitions

The string operations that are defined here have some points in common. In particular, they are uniform in their handling of the storage types, both for arguments and for results. Therefore, the following general conventions can be adopted for use in the definitions.

ARGUMENTS

As the first step in the interpretation of an operation, the storage types of the arguments are checked. In the case of string operations, there are not many possibilities for the storage types, and so the requirements can be indicated by using special symbols for the operands or arguments. For example:

(B1 & B2)

means that both operands of the "and" operator should be bit strings.

The complete conventions for the symbols used for the string operations are:

B, B1, B2, ...	The value of the argument should have string-type attribute 'bit'.
C, C1, C2, ...	The value of the argument should have string-type attribute 'char'.
S, S1, S2, ...	The value of the argument should have string-type attribute 'bit' or 'char'.
I, J, K, ...	The value of the argument should be arithmetic. It is converted to 'fixed bin(24,0)' unless it already has that storage type.

Some remarks on the significance of these conventions are necessary:

- In some cases, two string operands must agree in their lengths. In such cases, blank characters or zero bits are added at the right end of the shorter string until its length is equal to that of the other string.
- Arguments for which the symbol is I, J, K, ... either give a position in a string or the length of a string. The value 24 is used as the number-of-digits for all such arguments because a 24-bit integer is the smallest that can accommodate the length of the longest possible string in Multics PL/I.

Concatenate Operations

The simplest operation on string values is concatenation. For this purpose, PL/I has two built-in operations:

- The concatenate operation, which yields a string that starts with the first operand string and continues with the second.
- The 'copy' function, which concatenates a given string with itself a given number of times.

These two operations nicely reflect the balance in the set of built-in operations of PL/I. The concatenate operator is fundamental and cannot be expressed in terms of simpler operations. In contrast, the 'copy' function could easily be programmed by the user in terms of the concatenate operation, but the programmed form would be much less efficient than the built-in 'copy' function.

CONCATENATE OPERATOR

A parenthesized operator expression for the operator has the form:

```
( S1 || S2 )
```

The result is the concatenation of S1 and S2. For example:

```
( "abc"||"de" ) = "abcde"  
( "abcde"||"" ) = "abcde"  
( "abcde"||"ØØ" ) = "abcdeØØ"
```

If the values of the operands, S1 and S2, are both bit strings, the result is a bit string:

```
( "011"b||"1101"b ) = "0111101"b
```

'copy' FUNCTION

A reference to the function has the form:

```
copy(S,I)
```

The result is the concatenation of I copies of S. For example:

```
copy("abc",2) = "abcabc"  
copy("abc",0) = ""
```

If S has a bit-string value, the result is a bit-string value:

```
copy("110"b,2) = "110110"b  
copy("110"b,0) = ""b
```

Substring Operations

There are five built-in operations for obtaining or locating a substring of a given string. They are:

- The 'substr' function, which yields a substring of the given string that begins at a given position and has a given length.
- The 'index' function, which yields the position in the given string of an occurrence of another given string.
- The 'before' and 'after' functions, which yield a substring of the given string that occurs before or after an occurrence of a second given string.
- The 'decat' function, which is a generalization of the 'before' and 'after' functions.

The 'substr' and 'index' functions are fundamental to any string processing. The remaining functions are used primarily in advanced string processing, such as compiling and command-string processing.

'substr' FUNCTION

A reference to the function has one of the forms:

```
substr(S,I,J)
substr(S,I)
```

The result is the substring that begins with the Ith character and has length J. For example:

```
substr("abcde",1,4) = "abcd"
substr("abcde",3,1) = "c"
substr("abcde",3,0) = ""
```

If J is not given, the substring of S begins with the Ith character and continues to the end of S:

```
substr("abcde",1) = "abcde"
substr("abcde",3) = "cde"
```

Two kinds of errors are possible in the use of 'substr', and both cause the 'stringrange' condition to occur. First, it is an error to use a negative value for J (the length of the substring). Thus:

```
substr("abcde",3,-2) = ? ('stringrange' condition)
```

Second, it is an error to attempt to refer to a character that is beyond the end of the given string. Thus:

```
substr("abcde",3,4) = ? ('stringrange' condition)
substr("abcde",0,2) = ? ('stringrange' condition)
```

There is an exception to this rule: the character after the end of S can be referred to provided it is not used in the substring; this can only happen if the requested string has length 0. Thus:

```
substr("abcde",6,0) = ""
substr("abcde",6) = ""
```

The pseudo-variable named 'substr' is described later, in Section X, "Value Assignment."

'index' FUNCTION

A reference to the function has the form:

```
index(S1,S2)
```

The result is a 'fixed bin(24,0)' value that is the position of the beginning of the leftmost occurrence of S2 in S1. For example:

```
index("abcde","d") = 4
index("abcde","de") = 4
index("abcde","abcde") = 1
index("101110"b,"111"b) = 3
```

If S2 is not contained in S1, then the result is zero; thus:

```
index("abcde","bxd") = 0
index("", "d") = 0
index("101110"b,"010"b) = 0
```

If S2 is the null string, the result is also zero; thus:

```
index("abcde","") = 0
```

The function can be used both to locate a substring and to determine whether a substring is present. For some arguments, 'index' is related to 'substr', as follows:

```
I = index(S,substr(S,I))    for 0 < I <= length(S)
```

To illustrate this identity, suppose S is "abcde" and I is 2, then:

```
2 = index("abcde",substr("abcde",2))
2 = index("abcde","bcde")
2 = 2
```

'before' FUNCTION

A reference to the function has the form:

```
before(S1,S2)
```

The result is the substring of S1 that is before the leftmost occurrence of S2 in S1. For example:

```
before("abcde","bc") = "a"
before("abcde","abc") = ""
before("abcde","e") = "abcd"
before("abcde","abcde") = ""
```

If necessary, a null string is assumed at the beginning of S2; therefore, nothing is before a null string:

```
before("abcde","") = ""
```

If there is no occurrence of S2 in S1, then an occurrence is assumed at the end of S1; therefore, all of S1 is before S2:

```
before("abcde","bxd") = "abcde"
before("", "a") = ""
```

If both arguments yield bit strings, then the result is a bit string:

```
before("011101"b,"11"b) = "0"b
```

'after' FUNCTION

A reference to the function has the form:

```
after (S1,S2)
```

The result is the substring of S1 that is after the leftmost occurrence of S2 in S1. For example:

```
after("abcde","bc") = "de"
after("abcde","abc") = "de"
after("abcde","e") = ""
after("abcde","abcde") = ""
```

If necessary, a null string is assumed at the beginning of S1 (as with 'before'); therefore, all of S1 occurs after the null string:

```
after("abcde","") = "abcde"
```

If there is no occurrence of S2 in S1, an occurrence is assumed at the end of S1 (as with 'before'); therefore, none of S1 is after S2:

```
after("abcde","bxd") = ""
```

If both arguments yield bit strings, then the result is a bit string:

```
after("1110111"b,"0"b) = "111"b
```

The 'before' and 'after' functions are closely related. The expression

```
before(S1,S2)||after(S1,S2)
```

always yields a copy of S1 from which the leftmost substring that matches S2 has been deleted; for example:

```
before("abcde","cd")||after("abcde","cd") = "abe"
before("abcde","cx")||after("abcde","cx") = "abcde"
```

'decat' FUNCTION

A reference to the function has the form:

```
decat(S1,S2,B)
```

The result is a string that is "deconcatenated" from S1 in a relatively complicated way. The third argument, B, is converted to a bit string of length three. The result string of the 'decat' function is thought of as a sequence of three substrings, and each substring is controlled by one of the bits in the value of B, as follows:

- If the first bit is one, then the first substring is 'before(S1,S2)'; otherwise, it is the null string.
- If the second bit is one and S2 is contained in S1, then the second substring is S2; otherwise, it is the null string.
- If the third bit is one, then the third substring is 'after(S1,S2)'; otherwise, it is the null string.

Thus, for example, for any S1 and S2:

```
decat(S1,S2,"101"b) = before(S1,S2)||after(S1,S2)
decat(S1,S2,"000"b) = ""
decat(S1,S2,"100"b) = before(S1,S2)
```

If S2 occurs in S1:

```
decat(S1,S2,"110"b) = before(S1,S2)||S2
decat(S1,S2,"111"b) = before(S1,S2)||S2||after(S1,S2)
```

But if S2 does not occur in S1:

```
decat(S1,S2,"110"b) = before(S1,S2)
decat(S1,S2,"111"b) = before(S1,S2)||after(S1,S2)
```

Some specific examples are:

```
decat("abcde","d","000"b) = ""
decat("abcde","d","001"b) = "e"
decat("abcde","d","010"b) = "d"
decat("abcde","d","011"b) = "de"
decat("abcde","d","100"b) = "abc"
decat("abcde","d","101"b) = "abce"
decat("abcde","d","110"b) = "abcd"
decat("abcde","d","111"b) = "abcde"
```

The primary purpose of the 'decat' function is to provide eight functions under a single name. A secondary purpose is to allow convenient selection of a function at execution time; this is done by using a variable expression for B.

Relational, Length, and Reverse Operations

After the concatenation and substring operations have been considered, only a few built-in operations remain that can apply to both character and bit strings. They are:

- The relational operators, which includes '=', '<', and six other similar operators. Only the application of these operators to string values is defined here.
- The 'length' function, which yields the current length of a given string.
- The 'maxlength' function, which yields the maximum length of a given string.
- The 'reverse' function, which yields a string that contains the characters or bits of the given string but in reverse order.
- The 'ltrim' and 'rtrim' functions, which trim unwanted characters, such as white space, from the left and right side, respectively, of character strings.

The relational operators (as applied to strings) are primarily used in advanced string processing, especially in sorting text data. The 'length' function is fundamental to any string manipulation. The 'reverse' function is used in advanced string processing.

RELATIONAL OPERATORS FOR STRING VALUES

A parenthesized expression for a relational operator has the form:

(S1 op S2)

where the relational operator, op, is one of the following:

= (is equal to)
< (is less than)
> (is greater than)
<= (is less than or equal to)
>= (is greater than or equal to)
^= (is not equal to)
^< (is not less than)
^> (is not greater than)

The relational operators can apply to many kinds of values, as follows:

- If both operands yield string values and neither is declared with a numeric picture attribute, then the expression is a string comparison and is described here.
- If either operand yields an arithmetic value or a character string declared with a numeric picture attribute, then the expression is an algebraic comparison and is described earlier in this section under "Relational Operators for Arithmetic Values."
- If both operands yield address values, then the expression is an address comparison and is described later in this section under "Relational Operators for Address Values."

The result of a comparison expression is "1"b or "0"b depending on whether the comparison is true or false. When the operand values have the same length and type, they can be equal only if they are identical. For example:

```
( "abcde" = "abcde" ) = "1"b
( "abcde" = "abcdx" ) = "0"b
( "abcde" ^= "abcdx" ) = "1"b
( "1101"b = "1101"b ) = "1"b
```

When the operand values have different lengths, characters or bits are added to the right end of the shorter operand value. Blank characters are added to a character string; zero bits are added to a bit string. For example:

```
( "abcde" = "abc" ) = ( "abcde" = "abc" ) = "0"b
( "11"b = "111"b ) = ( "110"b = "111"b ) = "0"b
```

Because of the padding of a short operand value, operand values that are not originally identical can satisfy the '=' operator; for example:

```
( "abc" = "abc" ) = ( "abc" = "abc" ) = "1"b
( "11"b = "110"b ) = ( "110"b = "110"b ) = "1"b
```

The result of a comparison expression whose operand includes '<' or '>' depends on the collating sequence. The collating sequence is a character string in which each possible character occurs exactly once. A character is less than another character if the first occurs before the second in the collating sequence. Similarly, a character is greater than another if the first occurs after the second. The collating sequence is an unalterable part of the definition of Multics PL/I, but it can vary from one implementation of PL/I to another.

The full collating sequence is given later in this section, under "The 'collate' Function". For the examples that are given here, an abbreviated version of the collating sequence is sufficient:

... (blank) ... 0123456789 ... abcdefghijklmnopqrstuvwxyz ...

Observe that, for example, 'c' is less than 'm' and, for another example, 'c' is greater than the blank character.

Examples of "less than" and "greater than" comparison expressions are:

```
( "c" < "m" ) = "1"b
( "c" ^< "m" ) = "0"b
( "c" <= "m" ) = "1"b
```

When the operand values are of length greater than one and are not identical, they are examined from left to right until a difference is found; the first differing character position is the sole basis for the comparison. For example:

```
( "abcde" < "abcxy" ) = ( "d" < "x" ) = "1"b
( "abcde" > "vwxyz" ) = ( "a" > "v" ) = "0"b
```

As already noted, operands of different lengths are adjusted so that their lengths agree; for example:

```
( "abc" < "abcde" ) = ( "abc" < "abcde" )
                      = ( " " < "d" ) = "1"b

( "000" < "0000" ) = ( "000" < "0000" )
                      = ( " " < "0" ) = "1"b
```

When bit strings are compared, '0' is less than '1'; thus:

```
( "0"b < "1"b ) = "1"b
( "101101"b > "101111"b ) = ( "0"b > "1"b ) = "0"b
( "1"b < "100"b ) = ( "100"b < "100"b ) = "0"b
```

'length' FUNCTION

A reference to the function has the form:

```
length(S)
```

The result is a 'fixed bin(24,0)' value that is the length of the string that is the value of S. For example:

```
length("abcde") = 5
length("") = 0
length("101010"b) = 6
```

Note that for a varying string the length is different from the maximum length. Suppose the variable 'x' is declared by:

```
del x char(10) varying;
```

and has the value "abc". Then:

```
length(x) = 3
```

rather than 10.

'maxlength' FUNCTION*

A reference to the non-Standard function has the form:

```
maxlength(S)
```

The result is a 'fixed bin(24,0)' real value that is the maximum length S can attain. This built-in gives the same result as the 'length' built-in except in cases where S is a varying bit-string or a varying character-string. For example:

```
del x char(20) varying;
x = "abc";
```

then,

```
length(x) = 3
maxlength(x) = 20
```

'reverse' FUNCTION

A reference to the function has the form:

```
reverse(S)
```

The result is a string which is the reverse of the value of S. For example:

```
reverse("abcde") = "edcba"  
reverse("a") = "a"  
reverse("") = ""  
reverse("11101"b) = "10111"b
```

Some of the built-in operations perform a left-to-right search of a string. When a right-to-left search is required, the 'reverse' function can be used to adjust the given string. The following example finds the position of the first occurrence of 'a' from the right end of the given string:

```
length("abacad") - index(reverse("abacad"),"a") + 1  
= 6 - index("dacaba","a") + 1  
= 6 - 2 + 1  
= 5
```

Note that this expression does not return zero if 'a' is not found in the given string, unlike the 'index' built-in.

'ltrim' FUNCTION*

A reference to the non-Standard function has the form:

```
ltrim(S, C) or ltrim(S)
```

The result is a string (S) whose value is trimmed on the left of all characters given in C. Characters are deleted until a character occurs that is not given in C, then the function stops. For example:

```
ltrim("5729 this example", "0123456789") = " this example"
```

If C is not specified, the value of C is a blank character. For example:

```
ltrim(" abcdef ") = "abcdef "
```

'rtrim' FUNCTION*

A reference to the non-Standard function has the form:

```
rtrim(S, C) or rtrim(S)
```

The result is a string whose value is trimmed on the right of all characters given in C. Characters are deleted until a character occurs that is not given in C, then the function stops. For example:

```
rtrim("abc123", "321") = "abc"
```

If C is not specified, the value of C is a blank character. For example:

```
rtrim(" abcdef ") = " abcdef"
```

This page intentionally left blank.

Bit-String Operations

Certain built-in string operations can be applied only to bit strings; they are:

- The logical operators, which perform the "not", "and", and "inclusive or" operations of Boolean logic.
- The 'bool' function, which is a generalization of the three logical operators and which can perform any of the 16 operations of Boolean logic.

The logical operators are used in all programming applications, especially when the operands are one-bit strings produced by tests of data. The 'bool' function is used in occasional advanced applications, especially for masking of bit strings.

LOGICAL OPERATORS

A parenthesized expression for a logical operator has one of the forms:

(~ B2)	(not)
(B1 & B2)	(and)
(B1 B2)	(inclusive or)

The result is a bit-string value. When both operands are a bit string of length one, the result is given by one of the following examples:

(~ "0"b) = "1"b	
(~ "1"b) = "0"b	
("0"b & "0"b) = "0"b	("0"b "0"b) = "0"b
("0"b & "1"b) = "0"b	("0"b "1"b) = "1"b
("1"b & "0"b) = "0"b	("1"b "0"b) = "1"b
("1"b & "1"b) = "1"b	("1"b "1"b) = "1"b

When operand strings are longer than one bit, the rules just given are applied to the individual bits, as follows. For the "not" operation, the rule is applied to the first bit of the operand to produce the first bit of the result; and so on, for the other bit positions:

(~ "1110001"b) = "0001110"b

For the "and" and "or" operations, the rules are applied to the first bit of the first operand and the first bit of the second operand to produce the first bit of the result; and so on for the other bit positions:

("1100"b & "1001"b) = "1000"b
("1100"b | "1001"b) = "1101"b

When one operand is shorter than the other, zero bits are added to the right end of the shorter operand; thus:

("1"b & "0011"b) = ("1000"b & "0011"b) = "0000"b

'bool' FUNCTION

A reference to the function has the form:

bool(B1,B2,B3)

The result is a bit string that is produced by applying to B1 and B2 and operation specified by B3. The third argument, B3, is converted to a bit string of length four. Suppose the bits in B3 are b1, b2, b3, and b4. Then if B1 and B2 are both bit strings of length 1, all possible results are given by the following examples:

```
bool( "0"b, "0"b, B3 ) = b1
bool( "0"b, "1"b, B3 ) = b2
bool( "1"b, "0"b, B3 ) = b3
bool( "1"b, "1"b, B3 ) = b4
```

As a specific example, suppose B3 is "0001"b; then:

```
bool( "0"b, "0"b, "0001"b ) = "0"b
bool( "0"b, "1"b, "0001"b ) = "0"b
bool( "1"b, "0"b, "0001"b ) = "0"b
bool( "1"b, "1"b, "0001"b ) = "1"b
```

Comparison of these equations with the definition of the '&' operator given earlier suggests that they are the same; indeed:

bool(B1,B2,"0001"b) = (B1 & B2)

The following table gives all possible results of operations performed on B1 and B2:

<u>B4</u>	<u>Name</u>	<u>Result</u>
0000	clear	all zeroes
0001	and	B1 & B2
0010		B1 & ^B2
0011	move B1	B1
0100		^B1 & B2
0101	move B2	B2
0110	xor	(B1&^B2) (^B1&B2)
0111	or	B1 B2
1111	^clear	all ones
1110	^and	^(B1&B2) = (^B1 ^B2)
1101		^(B1&^B2) = (^B1 B2)
1100	invert B1	^B1
1011		^(^B1&B2) = (B1 ^B2)
1010	invert B2	^B2
1001	^xor	^((B1&^B2) (^B1&B2)) = (^B1 B2) & (B1 ^B2)
1000	^or	^(B1 B2) = (^B1&^B2)

Thus the 'bool' function provides a general way of defining logical operations.

When B1 and B2 are bit strings of length greater than one, they are operated upon on a bit-by-bit basis just as with the logical operators. For example:

bool("111"b,"101"b,"0001"b) = "101"b

When one operand is shorter than the other, zero bits are added to the right end of the shorter operand, just as with the logical operators; thus:

```
bool("1"b,"0011"b,"0001"b) = bool("1000"b,"0011"b,"0001"b) = "0000"b
```

Character-String Operations

Three built-in operations can be applied only to character strings; they are:

- The 'search' and 'verify' functions, which are used to find the beginning and end of a generalized substring of a given string.
- The 'translate' function, which is used to modify a string by the systematic replacement of certain characters by certain other characters.

'search' FUNCTION*

A reference to the non-Standard function has the form:

```
search(C1,C2)
```

The result is a 'fixed bin(24,0)' value that is the position in C1 of the leftmost occurrence of any character contained in C2. Sometimes C2 is a single character; for example:

```
search("+328.02",".") = 5  
search("alpha,beta,gamma",",") = 6
```

In other cases, more than one character is used in C2; thus:

```
search("a2*(beta-gamma)","+-*/()") = 3  
search("18.2344e-2","e+-") = 8
```

When C1 does not contain any character in C2, the result is zero:

```
search("+328.02",";") = 0  
search("abcde","") = 0
```

This function works for non-ASCII characters as well as ASCII.

'verify' FUNCTION

A reference to the function has the form:

```
verify(C1,C2)
```

The result is a 'fixed bin(24,0)' value that is the position of the first character of C1 that does not occur in C2. For example:

```
verify("alpha-beta","abcdefghijklmnopqrstuvwxy") = 6  
verify("1923.98e02","0123456789") = 5
```

When C1 contains only characters that are in C2, the result is zero:

```
verify("1923.98e02","+-0123456789.e") = 0
```

The example just given suggests the reason for the name "verify"; the example checks a given string (which appears to be a decimal constant) to "verify" that it contains only allowed characters.

Although the function is used for "verification" as just described, it is more often used to "search for the end" of a given substring. As such, it is a complement to the 'search' function, which is used to "search for the beginning" of a given substring. Suppose a procedure must be written to extract the leftmost substring that is a sequence of digits. Before writing the procedure, the programmer sketches it out as follows:

The given string might be:

```
"alpha*(238+beta)"
```

The 'search' function is used to locate the beginning of the desired substring:

```
search("alpha*(238+beta)","0123456789") = 8
```

Application of the 'substr' function gives:

```
substr("alpha*(238+beta)",8) = "238+beta)"
```

The 'verify' function is used to locate the end of the desired substring:

```
verify("238+beta)", "0123456789") = 4
```

Application of the 'substr' function gives:

```
substr("238+beta)",1,3) = "238"
```

which is the desired result. This function works for non-ASCII characters as well as ASCII.

'translate' FUNCTION

A reference to the function has one of the forms:

```
translate(C1,C2,C3)  
translate(C1,C2)
```

The result is a modification of C1 in which each character in C3 is replaced in C1 by the corresponding character in C2. For example:

```
translate("abcde","B","b") = "aBcde"  
translate("abcde","ABCDEFG","abcdefg") = "ABCDE"  
translate("28.923e-18","000000000","0123456789") = "00.000e-00"  
translate("a","ABCD","aaaa") = "A"
```

C2 and C3 should be strings of equal length; however, if C2 is shorter than C3, blanks are added to the right end of C2:

```
translate("abcde","B","bcde")
  = translate("abcde","B    ","bcde") = "aB    "
```

Both C2 and C3 should be given (the second form is not recommended); however, if C3 is not given, the collating sequence is assumed:

```
translate(C1,C2) = translate(C1,C2,collate9())
```

This function works for non-ASCII characters as well as ASCII.

Character-Set Operations

Three built-in string operations are relevant to the character set of Multics PL/I. They are:

- The 'collate' function, which yields the string that gives the collating sequence of Multics PL/I.
- The 'high' and 'low' functions, which yield a sequence of control characters that have a special purpose in the preparation of output.
- The 'collate9' and 'high9' functions, which assume the character-set contains all possible 9-bit bit patterns.

'collate' FUNCTION

A reference to the function has the form:

```
collate          collate()
```

The result is a character string that gives the collating sequence for the characters in the ASCII character set. If a given character occurs before a second given character in the collating sequence, then the first character is "less than" the second. In this way, the collating sequence defines the relational operators as they apply to character strings.

The collating sequence is not the same in every implementation of PL/I; one reason for this variation is that different implementations use different character sets. Multics PL/I uses the ASCII character set, and the collating sequence is the ASCII character set arranged according to the ASCII octal codes. The character with code 000 is first, the character with code 001 is next, and so on. It follows that the ASCII code for a character can be obtained by means of the 'index' function. For example, to obtain the decimal equivalent of the ASCII code for the character "a", write:

```
index(collate(),"a")-1 = 97      (octal code 141)
```

Similarly, to obtain the character whose decimal code is 97, write:

```
substr(collate(),97+1,1) = "a"
```

There are 128 characters in the ASCII character set and therefore the length of the collating sequence is 128. The sequence can be described in four parts, as follows:

1. The first part is 32 nonprinting control characters, as follows:

```
000  NUL   (null character)
001
...   (unused)
006
007  BEL   (alarm)
010  BS    (backspace)
011  HT    (horizontal tab)
012  NL    (new line = carriage return and line feed)
013  VT    (vertical tab)
014  NP    (new page = carriage return and form feed)
015  CR    (carriage return)
016  RRS   (red ribbon shift)
017  BRS   (black ribbon shift)
020
...   (unused)
036
037  EGM   (enter graphic mode)
```

2. The second part is the blank character.
3. The third part is the 94 printing characters, as follows:

```
! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9
: ; < = > ? @
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
[ \ ] ^ _ `
a b c d e f g h i j k l m n o p q r s t u v w x y z
{ | } ~
```

4. The fourth part is a single nonprinting control character, as follows:

```
177  PAD   (padding character)
```

If words are alphabetized by means of the relational operators, then the collating sequence determines the order of the words. The important features of the collating sequence are:

- The blank character comes before any printing character; therefore the following ordering results:

```
plan
plane
```

- Any digit comes before any letter. Therefore the following ordering results:

```
alpha
alpha2
alpha3
alphabet
```

- Each case of letters (upper or lower) appears in the collating sequence in uninterrupted alphabetical order. However, any uppercase letter occurs before any lowercase letter. Therefore the following ordering results:

```

Boston
Cambridge
alpha
beta

```

- The punctuation marks are scattered throughout the collating sequence. Therefore they must be given special consideration in alphabetizing.

'collate9' FUNCTION*

A reference to the non-Standard function has the form:

```
collate9          collate9()
```

The result is a character set composed of the Multics Extended Character Set, the first 128 characters of which are the ASCII character set. For information on the Multics Extended Character Set, see the MPM Reference Guide. The 'collate9' function is analogous to the standard collate built-in function except that the return value contains all possible 9-bit bit patterns rather than just the 128 ASCII characters, therefore, making the length 512. For example:

```
substr(collate9(),1,128) = collate()
```

'low' FUNCTION

A reference to the function has the form:

```
low(I)
```

The result is a character string composed of I "NUL" characters. The "NUL" character is the first character in the collating sequence.

'high' FUNCTION

A reference to the function has the form:

```
high(I)
```

The result is a character string composed of I "PAD" characters. The "PAD" character is the last character in the ASCII character set.

'high9' FUNCTION*

A reference to the non-Standard function has the form:

high9(I)

The result is a character string composed of I characters all of whose bits are one. This character is the last character in the Multics Extended Character Set. The 'high9' function is analogous to the standard high built-in function except that the return value contains all possible 9-bit bit patterns rather than just the 128 ASCII characters. For example:

high9(1) = all bits are ones

String Functions Defined Elsewhere

The following built-in functions yield string values but are defined elsewhere, as indicated, because they are specialized:

bit character string	unspec valid	See "Conversion Operations".
onchar onfield onfile	onkey onloc onsource	See "System Variable Operations".
date	time	See "System Variable Operations".

ADDRESS AND AREA OPERATIONS

The operations discussed here include operators for the comparison of address values, functions that have addresses as their values, and a single function that has an area as its value. All of these operations are considerably removed from the field of routine programming; they are certainly useful where they are needed and include some of the innovative features of PL/I; but they are rarely required outside of the storage management techniques that characterize advanced applications.

General Address Functions

There are five general functions and operations for the manipulation of address values; they are "general" in the sense that they are independent of the representation of address values in a particular implementation of PL/I. The functions are:

- The two relational operators '=' and '^=', which can be used for the comparison of address values
- The 'addr' function, which yields a pointer value that is equivalent to any given variable reference
- The 'null' and 'null0' functions, which yield the special null locative value for a pointer or an offset.

The relational operators for address values are useful whenever file values, local values, and so on, must be manipulated; such applications occur in intermediate and advanced applications. The other functions discussed here are used primarily in advanced applications that use based variables, list processing, and storage sharing.

RELATIONAL OPERATIONS FOR ADDRESS VALUES

A parenthesized operator expression with a relational operator has the form:

(A1 op A2)

where A1 and A2 must yield address values and the operator, op, is one of the following:

= (is equal to)
^= (is not equal to)

Relational operators can also apply to computational values, as described earlier under "Relational Operators for Arithmetic Values" and "Relational Operators for String Values".

The result of the relational expression is "1"b or "0"b, depending on whether the comparison is true or false. Recall that there are six types of address values, as follows:

label: the destination of a 'goto' statement
entry: the destination of a 'call' or a function reference
format: the object of a remote format reference
pointer: the "absolute" address of a storage unit
offset: the "relative" address of a storage unit
file: the address of a file-state block

With one exception, both arguments of an address relational operator must have the same type. The exception is the combination of a pointer value and an offset value; in this case, the offset value is converted to a pointer value.

'addr' FUNCTION

A reference to the function has the form:

addr(U)

where U must be a reference to a connected variable. A variable is connected unless it is an aggregate variable whose components are interleaved with the components of some other variable. The result is a pointer value to the storage unit that is designated by U. Suppose the following declaration applies:

```
dcl 01 A(2),  
    02 B float,  
    02 C dec(6,2);
```

Examples of the function follow:

addr(A) gives a pointer to storage for the entire array of structures
addr(A(2)) gives a pointer to the second element of the array of structures
addr(A(2).C) gives a pointer to a scalar component of 'A'.
addr(A.C) is invalid because 'A.C' designates an unconnected variable

The variable 'A.C' is unconnected because it is made up of two scalar variables, designated by 'A(1).C' and 'A(2).C', that are not adjacent in storage. According to Section VII, "Storage Management," the components of A occur in storage in the sequence:

A(1).B A(1).C A(2).B A(2).C

and thus 'A(2).B' is between the two components of 'A.C'.

If U is a major (level-one) 'controlled' variable reference that is not currently allocated, then

addr(U) = null

and the reference is valid. If U is any other variable that is not currently allocated, then the reference is invalid.

'null' FUNCTION

A reference to the function has one of the forms:

null null()

The result is the null value of data type 'pointer'. Suppose P1 is a pointer variable. Then the statement:

P1 = null;

assigns the null value to 'P1' and thus gives P1 a defined value that does not point to any storage unit. The variable can later be tested by a statement such as:

if P1=null then goto L2;

Observe that a pointer variable can be in one of the following states:

- If no value has been assigned to the variable, then the value of the variable is undefined and any reference to that value is invalid.
- If the null value has been assigned to the variable, then the value can be assigned to another variable or compared to another locative value; but it cannot be used as a locator-qualifier because it does not designate a storage unit.
- If a non-null value has been assigned to the variable, then the value of the variable can be used for any purpose appropriate for a locative value.

'nullo' FUNCTION*

A reference to the non-Standard function has the form:

nullo nullo()

The result is the 'offset' value null. An offset variable is set to null to indicate that it does not point to any storage unit. The usage of 'nullo' is similar to that described under "The 'null' Function". The results of the null and nullo built-in functions are considered equal, if compared.

Implementation-Dependent Address Functions

There are nine functions that manipulate a pointer value in a way that depends upon the Multics representation of pointers; these functions are not, of course, part of Standard PL/I. The functions are:

- The 'baseptr' function, which generates a pointer to the first word of a segment whose number is given
- The nonstandard 'ptr' function, which generates a pointer to the Nth word of a given segment
- The 'addrel' function, which generates a pointer to the Nth word after the word designated by a given pointer
- The 'baseno' and 'rel' function, which yield an integer that is the segment number or the word-offset number for a given pointer
- The 'stackframeptr' function, which yields a pointer to the stack frame of the current block
- The 'stackbaseptr' function, which yields a pointer to the base of the current stack segment of the block
- The 'codeptr' function, which yields pointers for an entry, label, or format value
- The 'environmentptr' function, which yields the activation record pointer for an entry, label, or format value

These functions should be used only under special circumstances. They are appropriate for systems programming that is closely related to the implementation of the Multics System itself.

'baseptr' FUNCTION*

A reference to this non-Standard function has the form:

baseptr(N)

where N must yield either:

an integer value of no more than 35 bits or a bit-string value of length 18

The result is the pointer value defined by:

ring number:	current ring number
segment number:	N
word offset:	0
bit offset:	0

Thus the resulting pointer designates the first word of the segment whose number is N.

NONSTANDARD 'ptr' AND 'addrel' FUNCTIONS*

References to these functions have the forms:

```
pointer(P,N)          ptr(P,N)
addrel(P,N)
```

where P must yield a scalar 'pointer' value and N must yield either:

an integer value of no more than 35 bits, or a bit-string value of length 18.

The result is the pointer value defined by:

```
ring number:          the ring number of P
segment number:       the segment number of P
word offset:          N, for the 'ptr' function
                       W+N, for the 'addrel' function, where W is the word
                       offset of P
bit offset:           0
```

Thus the result pointer designates the Nth word from the beginning of the segment into which P points (for the 'ptr' function) or the Nth word after the word to which P points (for the 'addrel' function).

There is another built-in function named 'ptr' in PL/I; it is described later, under "Conversion Operations", and is part of Standard PL/I. The two functions are distinguished by the data type of their second argument. For the standard function, the second argument is 'area'; for the nonstandard function, the second argument is arithmetic or 'bit'.

'baseno' AND 'rel' FUNCTIONS*

References to these non-Standard functions have the forms:

baseno(P)
rel(P)

where P must yield a scalar 'pointer' value. The result is a bit-string value of length 18 whose value is the bit-string representation of the segment number or the word offset, respectively, of P.

'stackframeptr' FUNCTION*

A reference to the non-Standard function has the form:

stackframeptr stackframeptr()

The result is a pointer value to the stack frame of the current block.

'stackbaseptr' FUNCTION*

A reference to the non-Standard function has the form:

stackbaseptr stackbaseptr()

The result is a pointer value to the base of the current stack segment of the block.

'codeptr' FUNCTION*

A reference to the non-Standard function has the form:

codeptr(X)

where X must yield an entry, label, or format value. The result R is a pointer value. If X is an entry value, then R is a pointer to the entry point identified by X. If X is a label value, then R is a pointer to the 'statement' identified by X. If X is a format value, then R is a pointer to the 'format statement' identified by X.

'environmentptr' FUNCTION*

A reference to the non-Standard function has the form:

environmentptr(X)

where X must yield an entry, label, or format value. The result R is the activation record pointer of X.

Area Function

There is one function for area values. It is:

- The 'empty' function, which gives the value of an area in which no variable is currently allocated

This function is used in advanced applications where special storage management techniques are used.

'empty' FUNCTION

A reference to the function has the form:

```
empty          empty()
```

The result is the empty value of data type 'area'. Suppose A is declared as an area variable; then the statement:

```
A = empty;
```

can be used to assign the empty value to 'A' and thus free all storage units currently allocated in 'A'.

ARRAY OPERATIONS

Each of the functions described here must have an array as its first operand, and it is in this sense that these are array functions.

Extent Functions

There are three functions that yield the values that describe a dimension of an array. They are:

- The 'lbound' and 'hbound' functions, which yield the bounds of a given dimension of an array
- The 'dimension' function, which yields the extent of a given dimension of an array

These functions are useful for operations on an array whose bounds are given by variable expressions. They are essential for determining bounds or extents of an array that is a procedure parameter with '*' extents; this case is discussed later, in Section XII, "Procedure Invocation."

'lbound' AND 'hbound' FUNCTIONS

References to these functions have the forms:

```
lbound(A,N)
hbound(A,N)
```

where A must yield an array value and N must yield a scalar value that can be converted to a 17-bit integer. The result is a 24-bit integer whose value is the lower bound or upper bound, respectively, of the Nth dimension of A. As a basis for examples, consider the program:

```
P:  proc;
    decl a2(j,-1:k+2,0:3) float controlled;
    ...
    j = 6;
    k = 20;
    allocate a2;
    ... (Computation #1)
    end;
```

As Computation #1 begins, the functions have the following values:

```
lbound(a2,1) = 1           hbound(a2,1) = 6
lbound(a2,2) = -1          hbound(a2,2) = 22
lbound(a2,3) = 0           hbound(a2,3) = 3
```

If N does not designate a declared dimension of A, a reference to either of these functions is invalid.

'dimension' FUNCTION

A reference to the function has the form:

```
dimension(A,N)           dim(A,N)
```

where A must yield an array value and N must yield a scalar value that can be converted to a 17-bit integer. The result is the extent of the Nth dimension of A. For any valid arguments, A and N:

```
dim(A,N) = hbound(A,N) - lbound(A,N) + 1
```

Special Array Functions

There are three functions for special operations on arrays. Each corresponds to a well-known mathematical operation, as follows:

- The 'sum' function, which performs the summation operation and is represented in mathematics by a capital sigma.
- The 'prod' function, which performs the product operation and is represented in mathematics by a capital pi.
- The 'dot' function, which performs the dot product operation and is represented in mathematics by a dot written between two vector names.

These functions are usually applied to 'float' values, and their use in that context is simple. However, they are also defined for 'fixed' values.

'sum' AND 'prod' FUNCTION

References to the functions have the forms:

```
sum(A)
prod(A)
```

where A must yield an array whose elements are arithmetic values. The result is a scalar value that is the sum or product of the elements of A. The data type of the result is the same as that of the argument, except for the precision of a fixed-point value. If A is 'fixed' with precision '(p,q)', then the precision of the result is:

```
(71,q)    for 'binary' base
(59,q)    for 'decimal' base
```

As a basis for discussion, consider the program:

```
P:  proc;
    dcl a3(2,3) float dec(5);
    dcl a4(3) fixed dec(6,2);
    ...
    a3 = 5;
    a3(1,3) = 10;
    a4(1) = 6;
    a4(2) = .5;
    a4(3) = .04;
    ... (Computation #1)
end;
```

As Computation #1 begins:

```
sum(a3) = 00035e0
sum(a4) = 0...06.54    (59 digits)

prod(a3) = 31250e0
prod(a4) = 0...0.12    (59 digits)
```

'dot' FUNCTION

A reference to the function has one of the forms:

```
dot(A1,A2,P)
dot(A1,A2,P,Q)
```

where A1 and A2 must each yield a one-dimensional array whose elements are arithmetic values, and where P and Q cannot be general expressions but instead must be decimal integer constants (Q can be signed). The precision of the converted values of A1 and A2 is given by:

<u>Reference</u>	<u>Scaling</u>	<u>Precision</u>
dot(A1,A2,P)	fixed	(P,0)
	float	(P)
dot(A1,A2,P,Q)	fixed	(P,Q)

The result is the dot product of A1 and A2; that is,

$$A1(m1)*A2(m2) + A1(m1+1)*A2(m2+1) + \dots + A1(m1+e)*A2(m2+e)$$

where

```
m1 = lbound(A1,1)
m2 = lbound(A2,1)
e  = dim(A1,1)
```

The operands must satisfy the condition:

$$\text{dim}(A1,1) = \text{dim}(A2,1)$$

CONVERSION OPERATIONS

With the exception of the special conversion functions, every conversion function given here can be interpreted by determining the data type of the target and then referring to Section IV, "Value Conversion," for a description of the required conversion. Thus although many examples of conversion are shown here, the rules for conversion of values are not given here.

The conversion functions of PL/I are both redundant and incomplete. There are three different ways to perform most, but not all, conversions; yet, for some important conversions, there is no way to perform the conversion that is acceptable to both Multics PL/I and Standard PL/I. This situation is discussed later in this section, under "Guidelines for Conversion Functions", and suggestions are given for the choice of a conversion function for various situations.

The designers of Multics PL/I took the view that a conversion between the three major types of computational value,

arithmetic,
character-string, and
bit-string

should be performed explicitly by means of a conversion function. Therefore, the compiler provides a warning message whenever such a conversion is performed implicitly. In compensation, to make explicit conversion easier, the designers added the nonstandard 'convert' function, whose interpretation is much simpler than that of the standard conversion functions.

Fundamental Conversion Function

There is one function that can perform all of the conversions between scalar values that are possible in PL/I. It is:

- The 'convert' function, which converts a given value to the data type of a given variable.

A simple and safe policy for conversion is to use only the 'convert' function for all explicit conversion. A more advanced policy is described under "Guidelines for Conversion".

'convert' FUNCTION*

A reference to the non-Standard function has the form:

convert(U,V)

where U must be a reference to a scalar variable and V must yield a scalar value. The result of the function reference is the value of V converted to the data type of U. The function can be used to establish any data type as the target for conversion, provided the conversion is valid in PL/I. The set of valid conversions is described earlier, in Section IV, "Value Conversion."

A specialized but important application of this function arises when a value of one major type must be assigned to a variable of a different major type. Suppose, for example, that the following declarations apply:

```
dcl alpha float;
dcl beta char(20);
```

Within the scope of these declarations, the assignment statement:

```
alpha = beta;
```

is marked by a warning message by the Multics PL/I compiler because it requires an implicit conversion between major types. On the other hand, the assignment statement

```
alpha = convert(alpha,beta);
```

is not marked because the conversion is explicit. Conversion between major types is usually considered to be important enough to merit an assignment statement of its own (rather than being performed with an expression); therefore, the usage just shown is common.

In order to consider the function in a general way, suppose the following declarations apply:

```
dcl fixdec62 fixed dec(6,2) based;
dcl S char(12);
```

Within the scope of these declarations:

```
convert(fixdec62,00028.3356) = 0028.33
convert(fixdec62,55528.3356) = (size)

convert(fixdec62,"-28") = 0028.00

convert(fixdec62,"-1111.001b") = -0015.12

convert(S,"123456789012") = "123456789012"
convert(S,"1234567890123") = (stringsize)

convert(S,100) = "XXXX100XXXXXXXX"

convert(S,-5.1749e-2) = "-5.1749e-002"
```

The only purpose of the variable reference that is the first argument of 'convert' is to supply a data type; the value of that variable is neither used nor set. The variable can be declared especially for use in 'convert' or it can be a variable that is used for other purposes as well. If the variable is especially declared, it should be 'based' so that no storage will be allocated for it.

The '*' means that this function is not part of Standard PL/I. A use of this function does not perform an operation that cannot be performed in Standard PL/I, but a program that uses 'convert' must be slightly changed to make it Standard. More is said of this under "Guidelines for Conversion Functions".

Conversion to Arithmetic Data Types

There are seven functions for conversion to an arithmetic data type, one for each of the attributes that are used in an arithmetic data type plus the 'imag' function. They are:

- The 'real' and 'complex' functions for mode conversion
- The 'fixed' and 'float' functions for scale conversion
- The 'binary' and 'decimal' functions for base conversion
- The 'precision' function for precision conversion

These functions can be used individually, but they should not be compounded, as in:

```
x = float(binary(y))
```

because the result is difficult to predict and often is not what is desired.

'real' FUNCTION FOR CONVERSION

A reference to this function has the form:

```
real(Z)
```

where Z must be 'complex'. The result is a 'real' value that is the value of the real part of Z. Except for the change in the mode attribute, the storage type of the result is the same as the storage type of Z. If the imaginary part of the value of Z is zero, then the result has the same mathematical value as Z, and the function therefore performs a conversion from 'complex' to 'real' mode. For example:

```
real(-6.0000e3+0.0000e0i) = -6.0000e3
```

```
real(.111000111e8b+.000000000e0bi) = .111000111e8b
```

If the imaginary part is not zero, the function does not perform a pure conversion operation, but rather an operation of complex arithmetic, which takes the real part of a complex number.

'complex' FUNCTION FOR CONVERSION

A reference to this function has the form:

complex(R,0)

where R must be 'real'. The result is a 'complex' value whose real part is the value of R and whose imaginary part is zero. The result has the same mathematical value as R, and the function (in this form) therefore performs a conversion from 'real' to 'complex'. Complex computations are usually performed with 'float' values; and for such values the storage type of the result is the same as the storage type of R except for the desired change of mode. Examples are:

complex(-6.0000e3,0) = -6.0000e3+0.0000e0i

complex(.111000111e8b,0) = .111000111e8b+.000000000e0bi

When the second argument of the 'complex' function is not zero, the function does not perform a pure conversion operations, but rather an operation of complex arithmetic as discussed earlier under "The 'complex' Function for Arithmetic".

'fixed' FUNCTION

A reference to this function has one of the forms:

fixed(X)
fixed(X,P)
fixed(X,P,Q)

where X must yield an arithmetic value and where P and Q cannot be general expressions but instead must be given as decimal integer constants (Q can be signed). The result is the value of X converted to a certain storage type. The storage type of the result is determined as follows:

<u>Reference</u>	<u>Data Type of X</u>	<u>Data Type of Result</u>
fixed(X)	<u>mode</u> fixed <u>base</u> (p,q) <u>mode</u> float <u>base</u> (p) char(m) bit(m)	<u>mode</u> fixed <u>base</u> (p,q) <u>mode</u> fixed <u>base</u> (p,0) real fixed dec(71,0) real fixed bin(63,0)
fixed(X,P)	...	as above, but with precision '(P,0)'
fixed(X,P,Q)	...	as above, but with precision '(P,Q)'

Examples are:

```
fixed(-0435.241)      = -0435.241
fixed(-0435.241,5,2) = -435.24
fixed(-0435.421,4,2) = (size)
fixed(-0435.421,4)   = -0435.

fixed(+000111000.000b,9,3) = +111000.000b

fixed(+000243.e2) = +024300.

fixed("-5.823e2")    = -0...582.      (59 digits)
fixed("-5.823e2",7,2) = -00582.30

fixed("110b",4,1) = 006.0
fixed("110"b,4,1) = 110.0b

fixed(-81143.e-2+68134.e-7i,3) = -811.+000.i
```

'float' FUNCTION

A reference to this function has one of the forms:

```
float(X)
float(X,P)
```

where X must yield an arithmetic value and where P cannot be a general expression but instead must be given as a decimal integer constant. The result is the value of X converted to a certain storage type. The scaling attribute of the result is determined as follows:

<u>Reference</u>	<u>Data Type of X</u>	<u>Data Type of Result</u>
float(X)	<u>mode</u> <u>fixed</u> <u>base(p,q)</u>	<u>mode</u> <u>float</u> <u>base(p)</u>
	<u>mode</u> <u>float</u> <u>base(p)</u>	<u>mode</u> <u>float</u> <u>base(p)</u>
	char(m)	real float dec(59)
	bit(m)	real float bin(63)
float(X,P)	...	as above, but with precision '(P)'

Examples are:

```
float(-89241e3)      = -89241e3
float(-89241e3,3)   = -892e5

float(.111000111000e4b,12) = +.111000111000e4b

float("002000000")  = 0...200e0      (59 mantissa digits)
float("002000000",5) = 00200e0
```

'binary' FUNCTION

A reference to this function has one of the forms:

binary(X)	bin(X)
binary(X,P)	bin(X,P)
binary(X,P,Q)	bin(X,P,Q)

where X must yield an arithmetic value and where P and Q cannot be general expressions but instead must be given as decimal integer constants (Q can be signed). The result is the value of X converted to a certain storage type that is determined as follows:

<u>Reference</u>	<u>Data Type of X</u>	<u>Data Type of Result</u>
bin(X)	<u>mode</u> fixed bin(p,q)	<u>mode</u> fixed bin(p,q)
	<u>mode</u> fixed dec(p,q)	<u>mode</u> fixed bin(p1,q1) p1 = ceil(p*3.32)+1 q1 = ceil(q*3.32) for q >= 0 q1 = -ceil(-q*3.32) for q < 0
	<u>mode</u> float bin(p)	<u>mode</u> float bin(p)
	<u>mode</u> float dec(p)	<u>mode</u> float bin(p1) p1 = ceil(p*3.32)
	char(m)	real fixed bin(71,0)
	bit(m)	real fixed bin(71,0)
bin(X,P)	...	as above, but with precisions '(P,0)' for a 'fixed' result and '(P)' for a 'float' result
bin(X,P,Q)	(this reference is not valid if X is 'float')	as above, but with precision '(P,Q)'

Examples are:

```
bin(-1110111.0111b) = -1110111.0111b
bin(-1110111.0111b,8) = -01110111b
bin(28.25) = 00011100.0100000b
bin(-.11101110111e8b) = -.11101110111e8b
bin("28.25",12,6) = 011100.010000b
bin("111000111"b,12,2) = 0111000111.00b
```

'decimal' FUNCTION

A reference to this function has one of the forms:

```
decimal(X)           dec(X)
decimal(X,P)        dec(X,P)
decimal(X,P,Q)      dec(X,P,Q)
```

where X must yield an arithmetic value and where P and Q cannot be general expressions but instead must be given as decimal integer constants (Q can be signed). The result is the value of X converted to a certain storage type that is determined as follows:

<u>Reference</u>	<u>Data Type of X</u>	<u>Data Type of Result</u>
dec(X)	<u>mode</u> fixed bin(p,q)	<u>mode</u> fixed dec(p1,q1) p1 = ceil(p/3.32) + 1 q1 = ceil(q/3.32) for q >= 0 q1 = -ceil(-q/3.32) for q < 0
	<u>mode</u> fixed dec(p,q)	<u>mode</u> fixed dec(p,q)
	<u>mode</u> float bin(p)	<u>mode</u> float dec(p1) p1 = ceil(p/3.32)
	<u>mode</u> float dec(p)	<u>mode</u> float dec(p)
	char(m)	real fixed dec(59,0)
	bit(m)	real fixed dec(59,0)
dec(X,P)	...	as above, but with precisions '(P,0)' for a 'fixed' result and '(P)' for a 'float' result
dec(X,P,Q)	(this reference is not valid if X is 'float')	as above, but with precision '(P,Q)'

Examples are:

```
dec(-0435.241)      = -0435.241
dec(-0435.241,5,3) = (size)

dec(1111.110000b)  = 015.75

dec(7.8100e0,7)    = 7.810000e0

dec("28.25",6,3)   = 028.250

dec("110.110b",4,3) = 6.750
```

'precision' FUNCTION

A reference to this function has one of the forms:

precision(X,P)	prec(X,P)
precision(X,P,Q)	prec(X,P,Q)

where X must yield an arithmetic value and where P and Q cannot be general expressions but instead must be given as decimal integer constants (Q can be signed). The result is the value of X converted to a certain storage type that is determined as follows:

<u>Reference</u>	<u>Data Type of X</u>	<u>Data Type of Result</u>
prec(X,P)	<u>mode</u> fixed <u>base</u> (p,q) <u>mode</u> float <u>base</u> (p) char(m) bit(m)	<u>mode</u> fixed <u>base</u> (P,0) <u>mode</u> float <u>base</u> (P) real fixed dec(P,0) real fixed bin(P,0)
prec(X,P,Q)	(this reference is not valid if X is 'float')	as above, but with precision (P,Q)

Examples are:

```
prec(18.7809,5,1) = 0018.7
prec(18.7809,5) = 00018
prec(-111000111.11b,10) = -0111000111b
prec(-8.2315e0,7) = -8.231500e0
```

Conversion to String Data Types

There are two functions for conversion to string data types. They are:

- The 'char' function, which converts a value to a character string.
- The 'bit' function, which converts a value to a bit string.

'character' FUNCTION

A reference to this function has one of the forms:

character(S)	char(S)
character(S,I)	char(S,I)

where S must yield an arithmetic value and I must be a scalar value that can be converted to a 24-bit integer. The result is a character string whose maximum length is determined as follows:

<u>Reference</u>	<u>Data Type of Result</u>
char(S)	char(*) nonvarying
char(S,I)	char(I) nonvarying

The '*' means "exactly enough characters to represent the value of S after it has been converted to a 'character' value". Examples are:

```

char("abcde")      = "abcde"
char("abcde",7)    = "abcde"
char("abcde",3)    = (stringsize)

char("11011"b)     = "11011"
char("11011"b,7)  = "11011"
char("11011"b,3)  = (stringsize)

char(+81993e6)     = "8.1993e+010"
char(+81993e6,14) = "8.1993e+010"
char(+81993e6,10) = (stringsize)

char(-0820)        = "820"
char(-0820,12)    = "820"
char(-0820,6)     = (stringsize)

char(.1011001...e7b,20) = "8.91250000e+001"

```

In the last example, "..." represents enough zeros to make the argument 'float(27)'.

'bit' FUNCTION

A reference to this function has one of the forms:

```

bit(S)
bit(S,I)

```

where S must yield an arithmetic value and I must yield a scalar value that can be converted to a 24-bit integer. The result is a bit string whose maximum length is determined as follows:

<u>Reference</u>	<u>Data Type of Result</u>
bit(S)	bit(*) nonvarying
bit(S,I)	bit(I) nonvarying

The '*' means "exactly enough characters to represent the value of S after it has been converted to a 'bit' value". Examples are:

```

bit("11011"b)      = "11011"b
bit("11011"b,7)    = "1101100"b
bit("11011"b,3)    = (stringsize)

bit("11011")       = "11011"b
bit("11012")       = (conversion)

bit(.1101111e5b)   = "11011"b
bit(18.983e0)      = "00000000000010010"b

bit(-065.29)       = "0001000001"b
bit(-065.29,14)   = "00010000010000"b

```

Conversion between Locative Data Types

There are two functions for conversion between the two locative data types. They are:

- The standard 'pointer' functions, which converts an offset value to a pointer.
- The 'offset' function, which converts a pointer value to an offset.

These functions are used when locative values are used in connection with 'area' values.

STANDARD 'pointer' FUNCTION

A reference to the function has the form:

pointer(X,A) ptr(X,A)

where X must yield a scalar offset value and A must yield a scalar area value. X must designate a storage unit that is currently allocated in the area A; and the result is a 'pointer' value that designates the same storage unit. Thus the function converts an 'offset' value to a 'pointer' value. This function has an 'area' value as its second argument, and is distinguished from the nonstandard 'pointer' function by that feature.

'offset' FUNCTION

A reference to the function has the form:

offset(P,A)

where P must yield a scalar pointer value and A must yield a scalar area value. P must designate a storage unit that is currently allocated in the area A; and the result is an 'offset' value that designates the same storage unit. Thus the function converts a 'pointer' value to an 'offset' value.

Special Conversion Functions

There are three special functions for conversion. They are:

- The 'string' function, which converts a packed aggregate of string values into a scalar string value.
- The 'unspec' function, which converts any value into a bit string that represents its actual representation in memory.
- The 'valid' function, which checks, after the fact, whether or not a given string can be assigned to a given pictured variable.

These functions are used in advanced applications that use storage sharing and special input/output.

'string' FUNCTION

A reference to the function has the form:

```
string(U)
```

The function reference is interpreted as follows:

- If U yields a scalar bit-string value, the result is U.
- If U yields a scalar value of any other computational type, the value is converted to 'char(*)' and the converted value is the result. The '*' means "exactly enough characters to represent the value of U when it has been converted to a 'character' value".
- If U yields an aggregate value whose components are unaligned, nonvarying, bit-string variables, the result is a scalar bit-string value that is the concatenation of all of the components of the value of U.
- If U yields an aggregate value whose components are unaligned, nonvarying, character-string or numeric-pictured variables, the result is a scalar character-string value that is the concatenation of all of the components of the value of U.

If none of these cases apply, then the reference is invalid. This is the only conversion function that can convert an aggregate value to a scalar value in an implementation-independent way. It is very useful in certain specialized applications. For example, consider the following program:

```
dcl 01 member unaligned,
    02 name char(30),
    02 address char(30),
    02 citystate char(30);
dcl smem char(90);
...
smem = string(member);
...
end;
```

In this program 'name', 'address', and 'citystate' can be manipulated as individual character strings, but when it is useful to have them as a single string, they can be assigned to 'smem' through the 'string' function. The assignment statement is equivalent to:

```
smem = name||address||citystate;
```

The 'string' function is closely related to string overlay defining which is discussed earlier, in Section VII, "Storage Management."

The pseudo-variable named 'string' is described later, in Section X, "Value Assignment."

'unspec' FUNCTION

A reference to the function has the form:

```
unspec(U)
```

where U must be a reference to a variable. The result is a bit-string value that is the Multics internal representation of the value of U, as described in Section VII, "Storage Management." Consider, for example, the program:

```
P:  proc;
    dcl 01 item unaligned,
        02 count fixed,
        02 code char(2),
        02 rate dec(2,1);
    ...
    count = 67;
    code = "CC";
    rate = -0.5;
    ...
end;
```

After the three assignment statements, the values of the 'unspec' function are:

```
unspec(count) = "000000000001000011"b
unspec(code)  = "xx1000011xx1000011"b
unspec(rate)  = "xx0101101xx0110000xx0110101"b

unspec(item)  = "000000000001000011xx1000011xx1000011
                xx0101101xx0110000xx0110101"b
```

The 'xx' is used for the two high-order bits of each character code because these bits are reserved in Multics. This example uses a variable that is 'unaligned', but (in contrast to the 'string' function) there is no restriction on the variable mentioned in the 'unspec' function. If this example were repeated without the 'unaligned' attribute, 'unspec(item)' would contain 108 bits instead of 63 bits.

One use of the 'unspec' function is as follows: The contents of a variable is converted to a bit-string by means of the 'unspec' function and the result is output as a record to some storage device; later, the string is input and is assigned back to a variable of exactly the same storage type by means of the 'unspec' pseudo variable. This use is a valid and standard use of PL/I, because it produces the same result regardless of the particular internal representation of a given implementation. Other uses, which test or manipulate the value of 'unspec' are not valid in Standard PL/I because they are, of course, implementation-dependent.

The pseudo-variable named 'unspec' is described later, in Section X, "Value Assignment."

'valid' FUNCTION

A reference to the function has the form:

```
valid(S)
```

where S must be a scalar pictured variable. The result is "1"b or "0"b, depending on whether or not the value contained in S can be edited into the picture declared for S. The function is used in connection with storage sharing to check to see if an invalid value has been assigned to the pictured variable by way of an equivalenced variable that does not have a picture. For example, consider the program:

```
P:  proc;
    dcl S1 pic"9999";
    dcl S2 char(4) based;
    dcl P pointer;
    P = addr(S1);
    ...
    P->S2 = "-500";
    ... (Computation #1)
end;
```

This program sneaks an invalid character-string value into the pictured numeric variable 'S1'; that is, an assignment is made to 'P->S2' when 'P' points to 'S1'. If the assignment had been written as:

```
S1 = "-500";
```

then the 'conversion' condition would have occurred. The 'valid' function can be used to check for such invalid assignments. The value of the function as Computation #1 begins is:

```
valid(S1) = "0"b
```

This indicates that the value of 'S1', a sign and three decimal digits is not consistent with the picture, four decimal digits.

Guidelines for Conversion Functions

For many data type conversions, there are three ways to effect the conversion:

- assignment to a variable of the desired storage type and related forms of implicit conversion
- use of one of the standard conversion functions; that is, the functions whose names coincide with data type attributes, such as 'fixed'
- use of the nonstandard 'convert' function

Suggestions for a choice from among these possibilities are given here.

If the target of the assignment is arithmetic, then special care must be exercised because the compounding of some standard conversion functions produces results which are undesirable. Therefore:

1. When a conversion can be effected by a single standard conversion function, that function should be used.
2. When a conversion can be effected by a mode function compounded with a single scaling, base, or precision function, then those functions should be used.
3. When neither Rule 1 nor Rule 2 applies:
 - a. If it is unlikely that the program will be transported from Multics to some other PL/I implementation, the 'convert' function should be used.
 - b. If transportation of the program is likely, the implicit conversion should be used. If Multics generates a warning message because of the implicit conversion, the message should be ignored.

If the target of the assignment is character-string or bit-string, then the standard functions should be used in preference to the 'convert' function because they produce satisfactory results without departing from the standard.

Some Multics programmers, who are not concerned about a departure from the standard, make consistent use of the 'convert' function to indicate all major conversions in a uniform way. In this case, the choice is a matter of programming style.

SYSTEM VARIABLE OPERATIONS

The PL/I interpreter maintains certain variables that cannot be accessed directly by a program. The purpose of most of the built-in functions described here is to provide a restricted form of access to these variables.

For example, there is a system variable that contains an encodement of the time of day, and this value can be retrieved by the 'time' built-in function but cannot be set by an ordinary PL/I program. For another example, there is a system variable for each 'print' file that is open that contains the current page number of the file, and this value can be retrieved by the 'pageno' function and set by the 'pageno' pseudo-variable.

Several functions included here do not access system variables, but those functions are highly specialized, nonstandard functions that do depend heavily on the implementation of PL/I.

System Counter Functions

There are six functions that take their values from counters maintained by the PL/I interpreter. They are:

- The 'lineno' and 'pageno' functions, which yield the appropriate integer values for a given 'print' file.
- The 'clock' and 'vclock' functions, which yield appropriate values in microseconds.
- The 'time' and 'date' functions, which yield appropriate string values.

These functions are of general interest.

'lineno' AND 'pageno' FUNCTIONS

References to these functions have the forms:

```
lineno(F)
pageno(F)
```

where F must yield a scalar file value that has the 'print' attribute. The result is a 35-bit integer that is the current line number or page number, respectively, of the file F. The value is obtained from the file-state block designated by F.

The pseudo-variable named 'pageno' is described later, in Section X, "Value Assignment."

'clock' AND 'vclock' FUNCTIONS*

References to these non-Standard functions have the forms:

```
clock      clock()
vclock     vclock()
```

The result is a fixed-point, binary, real value of precision (71,0). For 'clock' the result is the number of microseconds since 0000 hours January 1, 1901 Greenwich mean time. For 'vclock' the result is the number of microseconds of virtual CPU time used by the calling process.

'time' AND 'date' FUNCTIONS

References to these functions have the forms:

```
time          time()
date          date()
```

The result of the 'time' function is a character string of length 12 whose value is:

```
2 characters for the hour
2 characters for the minute
2 characters for the second
6 characters for the microseconds
```

The result of the 'date' function is a character string of length six whose value is:

2 characters for the year
2 characters for the month
2 characters for the day

For example:

time = 220318730211
date = 740703

is interpreted as:

10:03:18.730211 pm
July 3, 1974

Storage Management Functions

Three functions specifically designed for storage management are:

- The 'allocation' function, which yields the current number of allocations for a 'controlled' variable.
- The 'size' function, which yields the number of Multics words necessary to allocate a given variable.
- The 'currentsize' function, which yields the amount of storage presently occupied by the specified reference.

These functions are used in advanced applications where special storage management techniques are required.

'allocation' FUNCTION

A reference to the function has the form:

allocation(U) allocn(U)

where U must be a major (level one) controlled variable reference. The result is a 17-bit integer that is the number of generations of U that are currently allocated. Consider the program:

```
P:  proc;
    dcl a(100) char(20) controlled;
    ... (Computation 1)
    allocate a;
    ...(Computation 2)
    allocate a;
    ...(Computation 3)
    free a;
    free a;
    ... (Computation 4)
end;
```

The value of 'allocation(a)' is:

0, 1, 2, and 0

during computations 1, 2, 3, and 4, respectively.

'size' FUNCTION*

A reference to this non-Standard function has the form:

size(U)

where U must be a simple reference to a major (level-one) variable. The result is a 24-bit integer whose value is the number of words that would be necessary to allocate a storage unit for U at the time the 'size' function is evaluated. When U is declared with variable extent expressions, the value of this function depends on those expressions. For example, consider the following declarations:

```
dcl s char(2*i) controlled;
dcl 01 x based
      02 n fixed
      02 a(i+1 refer(n)) float;
dcl i fixed;
```

If i = 10, then:

```
size (s) = 5
size (x) = 12
size (i) = 1
```

'currentsize' FUNCTION*

A reference to this non-Standard function has the form:

currentsize(X)

where X must be a single reference to a major (level-one) variable. The result is a fixed-point, binary, real number of precision (19,0) whose value is the number of 36-bit words occupied by the generation of storage obtained by evaluating the reference X. Note that when X is a reference to a based variable with a 'refer option', this function returns a value that depends on the reference contained in the 'refer option' not on the 'expression' preceding the 'refer option'. For example:

```
dcl p ptr;
dcl 1 x based(p),
      2 n fixed,
      2 a(i+1 refer(n)) float;
dcl i fixed;
```

If i = 10, and the statement 'allocate' x is executed, then:

```
size(x) = 12
currentsize(x) = 12
```

Then if i is assigned the value 1:

```
size(x) = 3
currentsize(x) = 12
```

Resource Reservation Function

Two functions dedicated to the problem of resource reservation are:

- The 'stac' function, which places a bit string in a word if the word is zero, and does so in a single, indivisible operation.
- The 'stacq' function, which tests and sets a word of storage in a single operation.

These functions are used in advanced applications that require the coordination of Multics processes.

'stac' FUNCTION*

A reference to this non-Standard function has the form:

```
stac(P,B)
```

where P must yield a scalar pointer value and B must yield a scalar bit string of length 36. The following steps are performed:

1. The word designated by the pointer P is tested. The test succeeds if all bits of the word are zero.
2. If the test succeeded, the bit-string value B is assigned to the word designated by P.
3. If the test succeeded, the value of the function is "1"b; otherwise, it is "0"b.

The critical feature of the function is:

When this function is interpreted, steps 1 and 2 are performed by an indivisible operation of the hardware of Multics and cannot be interrupted.

The purpose of the function can be explained by describing a situation in which the function is not used and a serious error results. Suppose a certain word in storage, word x, is used to hold the reservation of a particular Multics resource, such as an input/output device, that can be used by only one process at a time. If the resource is in use, then it contains a nonzero bit-string that identifies the process that is using the resource; otherwise, the word contains a zero bit-string. Now suppose that both Process A and Process B seek to reserve the resource at nearly the same time. The following sequence of operations occur:

1. Process A tests word x and finds that it is zero. It is about to assign a value to word x when it is interrupted.
2. Process B tests word x, finds that it is zero, and assigns a value to it indicating that Process B has claimed the associated resource. Some time later, Process B is interrupted.
3. Process A resumes and assigns a value to word x claiming the associated resource.

As a result of this sequence, each process continues execution under the incorrect assumption that it has sole claim to the resource associated with word x. If the 'stac' function is used to set word x, this sequence cannot occur because there can be no interruption between the testing and setting of word x.

'stacq' FUNCTION*

A reference to this non-Standard function has the form:

stacq(L, A, Q)

where L must be an aligned, scalar bit-string variable of length 36 and A and Q must each be bit-strings of length 36 or less. The following steps are performed:

1. L and Q are compared.
2. If they are equal, the value of A is assigned to L and the value of the function is "1"b.
3. If they are not equal, no assignment is performed and the value of the function is "0"b.

The critical feature of this function is:

When this function is interpreted, all steps are performed by an indivisible operation of the hardware of Multics and cannot be interrupted.

The purpose of this function is the same as for the 'stac' function. The 'stacq' function simply allows the word tested to be other than zero for the test to succeed.

This page intentionally left blank.

'on' Condition Functions

When the PL/I processor detects an exceptional condition, it invokes an on-unit. This action is, in effect, an invocation of a procedure that does not have parameters; instead, the necessary data is communicated through system variables. These system variables are accessed by the 'on'-condition built-in functions.

For example, the exceptional condition 'key' occurs when an attempt is made to input a keyed record that does not exist. To support the processing of this condition, the key (a character string) for which no record could be found is placed in a system variable provided for the purpose. The on-unit that processes the condition can use the built-in function 'onkey' to retrieve the value of this system variable.

Every system variable associated with a condition is, in fact, a stack of variables and thus resembles a 'controlled' variable. Each time a condition occurs and is signalled, a new variable is allocated on each associated stack and its value is set. Each time the handling of a condition is complete, the most recent variable on each associated stack is freed and its value is lost. PL/I provides stacks for condition parameters because condition handling can be recursive; that is, a condition can occur while a previous occurrence of the same condition is being handled.

Before execution of a program begins, the PL/I processor allocates a variable for each of the system variables under discussion. This variable is set to a single blank character for the 'onchar', a zero for 'oncode', and a null string for other on-condition functions. When an 'on' condition is not currently being handled, the associated system variables yield the values just mentioned.

The 'on' condition built-in functions are meaningful only in the context of the 'on' conditions themselves; these are discussed later, under "Condition Handling". The whole subject is relevant only to intermediate or advanced programming applications, where the user cannot simply make use of the default handling of conditions provided by PL/I.

'onloc' FUNCTION

A reference to the function has the form:

```
onloc          onloc()
```

The value of this function is set whenever any condition occurs. When the condition is signalled, a character-string value that designates the most recently entered procedure block is placed on the stack associated with the 'onloc' function. Suppose, for example, the following program is executed:

```
P:  proc;
    dcl y float;
INV: proc(X) returns(float);
    dcl X float;
    return(1/X);
    end;
    ...
    Y = INV(0);
    ...
    end;
```

When the division is performed, the 'zerodivide' condition occurs and is signalled; and the value of the function is:

```
onloc = "INV"
```

When no condition is being handled, the value of the function is:

```
onloc = ""      (the null string)
```

'oncode' FUNCTION

A reference to the function has the form:

```
oncode          oncode()
```

The value of this function is set whenever any PL/I condition occurs. When the condition is signalled, a 35-bit integer that indicates the cause of the condition is placed on the stack associated with the 'oncode' function. When no condition is being handled, the value of the function is:

```
oncode = 0
```

Because the run-time subroutines that support the execution of PL/I programs are subject to modification and improvement, the list of error code values is subject to change and is not published in this document. Even when these codes are published, a program whose logic depends on a value of 'oncode' may not run properly on other implementations of PL/I or on future versions of Multics PL/I. Generally, the only valid use of the value of 'oncode' is as part of an error message.

'onkey' FUNCTION

A reference to the function has the form:

```
onkey          onkey()
```

The value of this function is set whenever a keyed input/output operation is in progress and one of the following conditions is signalled for that operation:

```
endfile      key      record      transmit
```

When one of these conditions is signalled, the key given in the input/output statement is placed on the stack associated with 'onkey'. Consider, for example, the statement:

```
read file(alpha) key("beta");
```

Suppose that file 'alpha' contains no record whose key is 'beta'. Then the key condition occurs and:

```
onkey = "beta"
```

When no 'key' condition is being handled, the value of the function is:

```
onkey = ""      (the null string)
```

'onfield' FUNCTION

A reference to the function has the form:

```
onfield          onfield()
```

The value of this function is set when the 'name' condition is signalled. The 'name' condition occurs during data-directed stream input/output when a name is encountered that is not mentioned in the list in the input statement. When the condition is signalled, the character string just extracted from the input stream is placed on the 'onfield' stack. Suppose, for example, the statement:

```
get file(sysin) data(alpha,beta,gamma);
```

is executed when the input stream is:

```
alpha=28.3,beta=61.4,gamma=19.2;
```

then the 'name' condition is signalled and the value of the function is:

```
onfield = "beta=61.4"
```

When no 'name' condition is being handled, the value of the function is:

```
onfield = ""      (the null string)
```

'onchar' AND 'onsource' FUNCTIONS

References to these functions have the forms:

```
onchar          onchar()
onsource        onsource()
```

The value of 'onsource' is the character string that is on the top of the stack associated with the 'onsource' function. The value of 'onchar' is a single character, the conversion character in the character string that is on the top of the stack associated with the 'onsource' function.

The 'onsource' and 'onchar' functions are associated with the 'conversion' condition. The 'conversion' condition occurs when an attempt is made to convert a character-string value to an arithmetic or pictured value and the attempt fails because the given character-string value has the wrong form. The conversion character is defined as follows:

- If the given character-string can be corrected only by changing some characters, then the conversion character is the leftmost character that must be changed in any correction of the given character-string.
- If the given character-string can be corrected simply by adding some characters at the end, then the conversion character is the last character of the given character-string.

When no 'conversion' condition is being handled, the values of the functions are:

```
onsource = ""
onchar   = " "
```

As a basis for an example of these functions, consider the following assignment statement:

```
x = float(y);
```

Suppose that 'y' is a character-string variable and its value is:

```
"-12300z5"
```

When the statement is executed and an attempt is made to evaluate the 'float' built-in function, the processor finds that the string "-12300z5" is not a valid representation of an arithmetic value. When the 'conversion' condition occurs:

```
onsource = "-12300z5"
onchar   = "z"
```

Now suppose that the value of 'y' is:

```
"-12300e+"
```

When the assignment statement is executed, the processor finds that the string is a valid beginning for a representation of an arithmetic value; for example, it could be corrected by adding the character '5'. Therefore:

```
onsource = "-12300e+"
onchar   = "+"
```

Finally, suppose that the value of 'y' is:

```
"-12300+5"
```

Once again, the processor finds that the given character string is not a valid representation for an arithmetic value. A human reader might say that the 'e' that separates the mantissa from the exponent is missing, so that the value of 'onchar' should be '+'. However, the PL/I processor recognizes that the representation can be corrected by adding an 'i' at the end to produce a representation of a 'complex' value. Therefore:

```
onsource = "-12300+5"  
onchar = "5"
```

The pseudo-variables 'onsource' and 'onchar' are described later, in Section X, "Value Assignment."

'onfile' FUNCTION

A reference to this function has the form:

```
onfile          onfile()
```

The value of this function is set whenever an input/output operation is in progress and one of the following conditions is signalled for the operation:

conversion	endfile	endpage	key
name	record	transmit	undefinedfile

When one of these conditions is signalled, the file-name of the file on which PL/I is operating is placed on the stack associated with the 'onfile' function. For example, consider the statement:

```
get file(alpha) list(X);
```

If the hardware fails to correctly transmit the value of X, the 'transmit' condition is signalled and the value of the function is:

```
onfile = "alpha"
```

Observe that the value of 'onfile' is not a file value; it is a character string that is the identifier that designates the file value. When none of the conditions listed above is being handled, the value of the function is:

```
onfile = ""      (the null string)
```

SECTION X

VALUE ASSIGNMENT

The assignment statement sets the value of a variable. This appears to be a simple action; however, two complications arise. First, the assigned value can have a different storage type from the target variable, and therefore an assignment statement sometimes requires a complicated conversion from one storage type to another. A thorough understanding of the rules given earlier, in Section IV, "Value Conversion," is required. Second, the order in which the actions of assignment are performed can, in certain special cases, affect the outcome of the assignment. These problems are discussed later in this section.

The description of assignment statements in this section begins with preliminary examples; these illustrate the rules in an informal way. Next, the form and interpretation of the assignment statement is defined in detail. Finally, the pseudo variables, which are special constructs associated with value assignment, are defined.

EXAMPLES OF ASSIGNMENT STATEMENTS

The following examples are given to illustrate the considerable variety of ways in which assignment statements can be used. Examples are given for each of the major data types: arithmetic, string, address, area, and array.

Arithmetic Assignment Statements

Most of the assignment statements in a typical program are short and simple. For example, consider:

```
i = m+1;
```

where both of the variable names are declared 'fixed'. This statement evaluates the right-hand-side expression 'm+1' and obtains an 18-bit integer. That value is converted to a 17-bit integer for assignment, and, if the high-order bit was one, the 'size' condition occurs. Finally, the converted value replaces the previous value of the target variable, 'i'.

An assignment statement can have a large right-hand-side expression and still be conceptually simple. For example, consider:

```
z = (x-1)**2 - (a+3*b)*(x-1) + (a-3*(b/c));
```

where all the variable names are declared 'float'. This statement is programmed entirely in floating-point binary and does not require any conversion operations. It is typical of engineering and scientific applications.

String Assignment Statements

The effect of a string assignment depends on whether or not the target variable is 'varying'. For example, consider:

```
S1 = "abc";
```

where 'S1' is declared 'char(5) nonvarying'. This statement sets 'S1' to "abc " (adding two blanks to the string) because that is the result of conversion to a 'char(5) nonvarying' target. In contrast, consider:

```
S2 = "abc";
```

where 'S2' is declared 'char(5) varying'. This statement sets 'S2' to "abc". If the assigned value is longer than the maximum size of the target, the 'stringsize' condition occurs, as described earlier, in Section IV, "Value Conversion."

A construct called a pseudo-variable can be used as the target of an assignment. A frequently used pseudo-variable is 'substr'. It allows a portion of a string variable to be changed. For example, consider:

```
substr(S,3,2) = "xx";
```

where 'S' is declared 'char(8) nonvarying'. This statement sets the third and fourth characters of S to "xx" and leaves the other characters unchanged.

The major types of computational values are arithmetic, character string, and bit string. A conversion between major types is allowed, but not recommended, in Multics PL/I. For example, consider:

```
X = "5";
```

where 'X' is declared 'dec(6,2)'. This statement assigns the value '0005.00' to 'X' (which is correct) in both Multics and Standard PL/I; but in Multics PL/I, the compiler marks the statement with a warning. This matter is discussed earlier, under "Conversion Operations" in Section IX, "Operations."

Address Assignment Statements

The assignment of address values is an advanced feature of PL/I. It is especially important in list-processing. For example, consider:

```
P = P->cell.next;
```

where 'P' is declared 'pointer' and 'cell' is a based structure whose member 'next' is declared 'pointer'. A statement of this form can be used to advance from one element of a list to the next element provided, of course, that the list structure is suitably defined.

Area Assignment Statements

An assignment statement can be used to copy the contents of one area variable into another. The effect is to copy the given area exactly as it appears, without any changes in the offsets or the current extent of the area. Consider the statement:

```
A1 = A2;
```

where 'A1' and 'A2' are declared 'area(100)' and 'area(200)', respectively. If the current extent of the 'area' value of 'A2' is greater than 100 (the maximum value of the 'area' variable 'A1'), then the 'area' condition occurs, as described earlier, in Section VII, "Storage Management."

Aggregate Assignment Statements

Aggregate values can be assigned to aggregate variables. For example, suppose two aggregate variables are declared as follows:

```
dcl 01 A(2),
    02 X1 float,
    02 X2 fixed;
dcl 01 B(3),
    02 alpha fixed,
    02 beta fixed;
```

Then the following assignment statement can be used:

```
A = B(1);
```

Observe that this statement requires the promotion of the value of the right-hand-side expression from the storage type:

```
01, 02 fixed, 02 fixed
```

to the storage type:

```
01 dim(2), 02 float, 02 fixed
```

The order in which the four scalar values are assigned to 'A', or the order in which conversions are performed, is not defined in PL/I.

FORM.OF ASSIGNMENT STATEMENTS

The assignment statement has the following form:

```
t,... = e;
```

where t is a target and e is the right-hand-side expression. The form 't,...' indicates that the statement can contain either a single target or a sequence of targets separated by commas. A target is either a variable reference or a pseudo-variable reference, and the right-hand-side expression is any expression. The pseudo-variables are built into PL/I, and they are:

```
real(ref)  
imag(ref)  
substr(ref,e1,e2)  
string(ref)  
unspec(ref)  
pageno(ref)  
onchar()  
onsource()
```

where ref is a variable reference appropriate to the pseudo-variable and e1 and e2 are expressions. The third argument, e2, of the 'substr' function can be omitted. The pseudo-variables are defined later in this section.

Targets

In most cases, the assignment statement has a single target, and that target is a variable reference. Thus:

```
x = (a+b**n)/n;  
A(i-F(z),phi+2) = 0;  
name(i).last = "Jones";  
g(alpha)->tab(k+3,m) = beta**3;
```

The statements just given illustrate the use of all four kinds of variable reference as target: simple, subscripted, structure-qualified, and locator-qualified. Examples of assignments with pseudo-variables as targets are:

```
substr(place,I-3,4) = "QQQQ";  
real(Z) = 2.8934e0;
```

Examples of assignment statements with multiple targets are:

```
k1,k2,k3 = 0;  
gamma, g(alpha)->tab(j,rho) = phi;  
mark, substr(part,4,3) = "mXT";
```

INTERPRETATION OF ASSIGNMENT STATEMENTS

An assignment statement is interpreted as follows:

1. Evaluate Right-Hand-Side Expression. Evaluate the right-hand-side expression and save its value.
2. Convert Value and Store Through Target. Process each target one by one, starting from the leftmost and (if there is more than one target) proceeding to the right. For each target, convert a copy of the value of the right-hand-side expression to the storage type of the target and then assign the converted value to the target.

The assignment of the converted value depends on the nature of the target, as follows:

- If the target is a variable reference, then the previous value of the designated variable is replaced by the assigned value. Since the assigned value has been converted to the same storage type as the target variable, the value fits exactly into the target variable.
- If the target is a pseudo-variable, the contents of some variable is changed; however, the variable is selected in an indirect way. The effect of assignment to a given pseudo-variable is given in the definition of that pseudo-variable, later in this section.

Special Restrictions

Assignment statements must satisfy several rather special restrictions. These restrictions, together with some explanation, are given here.

OVERLAPPING STRING TARGETS

Consider the following assignment statement:

S = T;

where both variables are declared 'char(500)'. The Multics implementation of PL/I copies a string value directly from the storage for 'T' into the storage for 'S' without using any intermediate storage. This is an efficient implementation; however, it can be followed only if certain troublesome cases, called overlapping string targets, are excluded from the language. The restriction given here excludes those cases.

The restriction on overlapping string targets requires the definition of the special string target. A given target is a special string target if all of the following statements are true:

- The given target is a variable reference or a reference to the 'substr' pseudo-variable.
- The given target occurs in a statement whose right-hand-side expression is either a scalar string reference or a reference to the 'substr' built-in function. In the latter case, the first argument of the function is a scalar string variable reference.
- The string type ('character' or 'bit') of the target and the right-hand-side expression must be the same, so that no conversion is necessary.

When a special string target designates all or part of the storage that is designated by the right-hand-side expression of the same statement, it is an overlapping special string target. Such a target is invalid.

This restriction is designed to permit the efficient execution of assignment statements that are of a relatively simple form. The reasoning behind the restriction is as follows:

1. There are some efficient methods for executing an assignment statement that has a special string target, as already noted.
2. The efficient methods sometimes produce invalid results when they are applied to overlapping special string targets.
3. It is not always possible for the PL/I compiler to distinguish between a special string target that is overlapping and one that is not.
4. Therefore, in order to permit the use of the efficient methods, the overlapping special string targets are excluded from PL/I.

As an example, consider the program:

```
P1: proc;
    dcl S char(100) var init("ABCDEFGH");
    dcl T char(100) var init("ABCDEFGH");
    substr(S,2,4) = substr(T,1,4);
    ...
end;
```

The assignment statement has a special string target that is not overlapping. It can be efficiently executed by assigning the first character of 'T' to the second character position of 'S', the second character of 'T' to the third character position of 'S', and so on. The result is "AabcdFGH", which is correct. Suppose, however, that the following assignment is used instead:

```
substr(T,2,4) = substr(T,1,4);
```

This assignment statement has an overlapping special string target and is invalid. To accomplish the indicated assignment, the programmer must write:

```
T1 = substr(T,1,4);  
substr(T,2,4) = T1;
```

where 'T1' is a suitably declared string variable name.

AREA ASSIGNMENTS

The application of an assignment statement to 'area' values is restricted to a statement of the following form:

Each of the targets is a reference to a scalar 'area' variable and the right-hand-side expression is a reference to a scalar 'area' variable or function.

Thus an 'area' value cannot be assigned as part of an aggregate value.

Order of Interpretation

A program must not depend on the order of the steps of the interpretation unless that order is explicitly stated in the definition of the language. Programs that depend on additional assumptions about ordering are invalid. The assignment statement is a construct in which this rule is especially important.

When an assignment statement assigns a scalar value to a single target, and when the right-hand-side expression does not invoke a function that has side effects, then no problems of ordering can arise. Most assignment statements are of this convenient kind; however, those that are not must be given special attention.

Each expression in an assignment statement is evaluated according to the ordering rules for expressions, as given earlier in Section VIII, "Expressions." In addition, the interpretation of an assignment statement is subject to the following ordering rules:

- The right-hand-side expression is evaluated before any value is assigned to a target.
- When an assignment statement has more than one target, the assignment of a value to a given target occurs before any expression (such as a subscript) in a subsequent target is evaluated.

Aside from these rules, the order of interpretation of an assignment statement is undefined.

The first of the ordering rules just given states that the generating expression is evaluated before a value is assigned to any target. That ordering can be important when an array value is assigned. Consider, for example, the statement:

```
A = A(2)*B;
```

where both 'A' and 'B' are declared 'dim(3) float'. Except for the ordering rule just given, one might suppose that an equivalent to this statement would be:

```
A(1) = A(2)*B(1);  
A(2) = A(2)*B(2);  
A(3) = A(2)*B(3);
```

This interpretation is not only incorrect but also undesirable, since it uses the old value of 'A(2)' as the multiplier for the first two elements of B and the new value for the last. Because of the ordering requirement, the correct interpretation uses the old value of 'A(2)' as the multiplier for all elements of B.

The definition of the assignment statement does not place a restriction on the time at which the location of a target variable begins. That is, although the assignment of a value to the target cannot occur until the evaluation of the generating expression is complete, there is no such restriction on the location of the target variable. In this connection, consider the statement:

```
A(i) = F(x);
```

where 'F' is the following user-defined internal procedure:

```
F:  proc(w) returns(float);  
    dcl w float;  
    ...  
    i = i+1;  
    ...  
    end;
```

The location of the target in the given assignment statement depends on whether the subscript in 'A(i)' is evaluated before or after the assignment 'i=i+1' is executed. Therefore, the given assignment statement is ambiguous and is invalid.

The two examples just given are typical, but they do not exhaust the set of interesting examples that could be given on the subject of ordering. The designers of PL/I specified ordering where they believed it was called for by common sense and conventional notation; and they left it out where it would have been an arbitrary rule. When the programmer is in doubt, he should assume that the ordering is unspecified; and then he should take some measure to provide a positive ordering, such as breaking a statement into a sequence of statements.

PSEUDO-VARIABLES

Some of the built-in functions can be used as targets; in that context, they are called pseudo-variables, not built-in function references. A pseudo-variable can be defined, for valid arguments, in terms of its corresponding built-in function reference. For example, consider the assignment statement:

```
real(Z) = real(Z);
```

Provided that 'Z' is a valid argument for the 'real' pseudo-variable and the 'real' built-in function, this statement changes nothing. That is, the generating expression yields the real part of 'Z', and then the pseudo-variable puts that same value back as the real part of 'Z'. All of the pseudo-variables behave in a similar manner.

A pseudo-variable can be used in only three contexts: as a target in an assignment statement (described in this section), as the target in a 'do' statement (as described later, in Section XI, "Program Flow"), and as a target in a list-directed or edit-directed 'get' statement (as described later, in Section XIV, "Stream Input/Output"). In each of these cases, a value is assigned to the pseudo-variable and the interpretation of the pseudo-variable then causes a value to be assigned to storage.

A pseudo-variable name must be declared 'builtin'. Observe that a given name can be used as both a pseudo-variable name and a built-in function name in the same block; the context of each reference determines whether it is a pseudo-variable or a built-in function reference.

A complete and independent definition for each pseudo-variable is given in what follows. Each definition gives the storage type to which the assigned value is converted, and then describes what the pseudo-variable does with that value.

'real' and 'imag' Pseudo-Variables

As pseudo-variables, the references have the same forms as the built-in function references, namely:

```
real(Z)
imag(Z)
```

The value assigned to the pseudo-variable is converted to the storage type of 'Z' except that the mode is 'real'; then the converted value is assigned to the real part or imaginary part, respectively, of 'Z'. For example, suppose the following declaration applies:

```
dcl alpha complex float dec(5);
```

and suppose 'alpha' has the value:

```
+5.0000e0-7.5000e0i
```

Then the assignment statement:

```
real(alpha) = -8.92;
```

sets the given value to `'-8.9200e0-7.5000e0i'`, while the assignment statement:

```
imag(alpha) = 0;
```

sets the given value to `'+5.0000e0+0.0000e0i'`.

'substr' as a Pseudo-Variable

As a pseudo-variable, the reference has the same forms as the built-in function reference, namely:

```
substr(S,I,J)          substr(S,I)
```

where `'S'` must be a string variable reference and `'I'` and `'J'` must be expressions whose values can be converted to 24-bit integers. The arguments can be aggregates; however, if `'I'` or `'J'` is an aggregate, then its aggregate type must be suitable for conversion to the aggregate type of `'S'`.

Consider, first, the interpretation of the pseudo-variable when all arguments are scalars. The data type of the pseudo-variable has the string type of `'S'` (`'character'` or `'bit'`), but has maximum length `J` and the attribute `'nonvarying'`. For example, consider:

```
substr(alpha,3,5) = "abc";
```

where `'alpha'` is declared `'char(10) varying'`. Here the data type of the target is `'char(5) nonvarying'`. The converted value, `"abc  "`, replaces that substring of `'alpha'` that begins with the third character of `'alpha'` and that is five characters long. If the assigned value had been more than five characters long, the `'stringsize'` condition would have occurred.

There are restrictions on the pseudo-variable. First, `'J'` must be zero or positive and `'I'` and `'J'` must specify a substring that lies entirely within the limits of the current value of the variable designated by `'S'`. Thus in the example given above, the current value of `'alpha'` must be at least seven characters long. When this restriction is not met, the `'stringrange'` condition occurs. Second, if `'S'` is a `'varying'` string, a value must be assigned to it before it appears in a `'substr'` pseudo-variable; otherwise, the current length of `S` would be undefined.

If any of the arguments of the pseudo-variable are aggregates, they are all converted to the aggregate type of `'S'`, and this aggregate type becomes the aggregate type of the pseudo-variable itself. The value of the generating expression is, in turn, promoted to this aggregate type before assignment. After conversion, the independent components are individually assigned through the `'substr'` pseudo-variable according to the rules for scalars.

'string' Pseudo-Variable

As a pseudo-variable, the reference has the same form as the built-in function reference, namely:

```
string(U)
```

where 'U' must designate one of the following:

- A scalar string variable that is 'nonvarying'.
- An aggregate variable whose components are all string variables of one type (either 'character' or 'bit', but not a mixture of 'character' and 'bit') and are all 'nonvarying' and 'unaligned'.

Such a variable is represented in storage as an uninterrupted sequence of characters or bits in a way that is independent of its aggregate type. Suppose the total number of characters or bits accommodated by the variable designated by 'U' is m. Then the pseudo-variable is interpreted as follows:

- First, the value assigned to the pseudo-variable is converted to 'bit(m)' or 'char(m)', depending on the type of 'U'.
- Second, if 'U' is a scalar, the converted value is assigned to it directly; otherwise, the converted value is assigned in such a way that the concatenation of the components of the aggregate variable 'U' are identical to the given converted value.

For example, consider the use of this pseudo-variable in the following program:

```
P:  proc;
      dcl 01 part unaligned,
          02 code,
          03 serial char(6),
          03 type char(2),
          02 descrip char(10);
      ...
      string(part) = "310-A6XXside";
      string(code) = "680-A3";
      ...
      end;
```

The effect of the two assignment statements on the value of 'part' is as follows:

	<u>first assignment</u>	<u>second assignment</u>
part.code.serial	"310-A6"	"680-A3"
part.code.type	"XX"	"??"
part.descrip	"side???????"	"side???????"

'unspec' Pseudo-Variable

As a pseudo-variable, the reference has the same form as the built-in function reference, namely:

```
unspec(U)
```

Suppose that the result of the built-in function reference 'unspec(U)' would yield a bit string of length m. (The interpretation of that built-in function reference is given under "Conversion Operations" in Section IX, "Operations.") The pseudo-variable is interpreted as follows:

- First, the value assigned to the pseudo-variable is converted to 'bit(m)'.
• Second, the converted value is assigned to 'U' in a way that the function reference 'unspec(U)' yields the given converted value.

A key point is that, for any variable reference 'x', the assignment statement:

```
unspec(x) = unspec(x);
```

leaves the value of 'x' unchanged. It then follows that the statements:

```
s = unspec(x);  
x = ... ;  
unspec(x) = s;
```

save the value of 'x' in 's', change the value of 'x', and then restore the value of 'x'.

The 'unspec' pseudo-variable is a way of interpreting the contents of raw storage (a sequence of bits) as a PL/I value. Together with the 'unspec' built-in function, it constitutes an escape from the machine-independence of PL/I and allows direct access to the storage of values in Multics words. Suppose, for example, it might be necessary to write a special PL/I procedure for converting a 'fixed' value to a 'float' value. In order to write such a procedure, it is necessary to prepare the mantissa and the exponent separately and then combine them into a single 'float' value by means of the 'unspec' pseudo-variable.

'pageno' Pseudo-Variable

As a pseudo-variable, the reference has the same form as the built-in function reference, namely:

```
pageno(F)
```

where 'F' must yield a scalar file value that has the 'print' attribute. The value assigned to the pseudo-variable is converted to a 35-bit integer. For example, suppose 'alpha' is declared as a file-name constant. Then the assignment statement:

```
pageno(alpha) = 100;
```

sets the page number counter for file 'alpha' to 100.

'onchar' and 'onsource' Pseudo-Variables

As pseudo-variables, the references have the same forms as the built-in function references, namely:

```
onsource          onsource()
onchar            onchar()
```

The value assigned to 'onsource' is converted (if necessary) to a character-string value and then replaces the value currently at the top of the stack that is associated with the 'onsource' built-in function. The value assigned to 'onchar' is converted (if necessary) to a 'char(1)' value and then replaces the conversion character (as defined below) in the character-string value currently at the top of the stack associated with the 'onsource' built-in function.

The 'onsource' and 'onchar' pseudo-variables are associated with the 'conversion' condition. The 'conversion' condition occurs when an attempt is made to convert a character-string value or pictured value to an arithmetic or bit-string value and the attempt fails because the given string has the wrong form. The conversion character is the leftmost character that must be changed as part of the correction of the given character-string value. A more detailed discussion of these matters is given earlier, under " 'on' Condition Functions" in Section IX, "Operations."

As an example of the use of 'onchar' as both a pseudo-variable and a built-in function, consider the following program:

```
P:  proc;
    dcl (sysin,sysprint) file;
    dcl x float;
    dcl conv cond;
    dcl onchar builtin;
    on conv
    begin
        put skip list("error in input");
        if onchar="1" then onchar = "1";
            else if onchar="0" then onchar = "0";
                else signal error;
    end;
    get list(x);
    put skip list(x**2);
end;
```

This program computes the square of a given number. The 'on' statement provides the program with a primitive form of error recovery: when the input value has either of the characters '1' or '0', the program prints a warning message and assumes that the corresponding digit was intended.

Suppose that the following input is supplied to this program:

123e1

The conversion of this input (which is a character-string value) for assignment to 'x' (which requires a 'float' value) proceeds as follows:

1. The first attempt at conversion fails. The first character, which is the letter 'l', is the conversion character. The 'conversion' condition is signalled. The 'on' unit prints a warning and replaces the conversion character with the digit '1'. Then the PL/I processor again attempts the conversion.
2. The second attempt also fails. The fifth character is also the letter 'l', and is handled in the same way as the first 'l'.
3. The third attempt at conversion succeeds because the given string is "123e1".

Although this example shows how the 'onchar' pseudo-variable works, it is a simple kind of recovery. A more sophisticated recovery procedure would use the 'onsource' built-in function to obtain the entire incorrect string, would analyze that string and make appropriate changes, and would use the 'onsource' pseudo-variable to replace the entire incorrect string with the corrected version.

SECTION XI

PROGRAM FLOW

As a program is executed, the PL/I interpreter passes from one statement to another; this part of program execution is the flow of control. There are seven kinds of flow of control in PL/I, as follows:

- Sequential flow is the execution of statements in the order in which they appear in the program. This kind of flow of control is used wherever some other kind is not explicitly called for.
- Conditional flow uses a test of current data values to determine whether or not a statement is executed. The 'if' statement is used for this purpose. A set of 'if' statements can be nested, one within the other, so it is possible to program a complicated case analysis using only 'if' statements.
- Iterative flow uses various conventions to execute a group of statements repeatedly. The 'do' statement is used to control the iteration. An index can be associated with the iteration.
- Transfer of control sends control to a specified statement. The 'goto' statement is used for this purpose. Because PL/I has both arrays of 'label' constants and unlimited 'label' variables, a rather general computation can be used to obtain the destination of the transfer.
- Block execution applies to a 'begin' block. A 'begin' block is used to declare variables in a restricted scope.
- Procedure invocation executes a block of statements as a closed subroutine. The 'call' statement or the function reference is used for procedure invocation. Provision is made for the transmission of arguments by either value or address, and procedures can be invoked recursively.
- Condition handling is used to process exceptional conditions that can occur during program execution, such as a division by zero or a transmission failure during input. The 'on', 'revert', and 'signal' statements are used for this purpose.

The first five kinds of flow of control are described in this section. The last two kinds require separate treatment and are described in Sections XII and XIII, "Procedure Invocation" and "Condition Handling," respectively.

SEQUENTIAL EXECUTION

When execution is sequential, statements are executed in the order in which they appear. However, when control reaches the end of a procedure, there is no "next statement" in the sequence; so the PL/I interpreter acts as if the next statement is

```
return;
```

and thus returns to the point at which the procedure was invoked.

The following constructs cause no action when they are encountered during sequential execution:

```
'procedure' blocks  
'format' statements  
'declare' and 'default' statements
```

Some of these constructs ('procedure' blocks and 'format' statements) cause action only when they are invoked by a proper form of reference. The remaining constructs never cause an action; they are present only to supply declaration information.

As an example of sequential execution, consider the following program:

```
P:  proc;  
    dcl (sysin,sysprint) file;  
    dcl (a,b,c) float;  
SQ: proc(x,y) returns(float);  
    dcl (x,y,z) float;  
    z = (x**2 + 5*y)/x;  
    return(z);  
    end;  
    get list(a,b);  
    c = 2*SQ(a,b);  
    put skip list(a,b,c);  
end;
```

Execution of the program is summarized as follows:

1. The 'procedure' statement, the two 'declare' statements, and the internal 'procedure' block are executed in sequence. The sequential execution of these constructs causes no action.
2. The input statement is executed and gets values for 'a' and 'b' from the input stream.
3. The assignment statement is executed. As part of its execution, it invokes the procedure 'SQ'; during that invocation, the statements in that procedure are executed sequentially.
4. The output statement is executed and prints the results.
5. Because the end of the external procedure has been reached, PL/I assumes a 'return' statement and returns to the command that invoked the external procedure.

'if' STATEMENT

An 'if' statement has one of the following forms:

```
if t then c1 else c2
```

```
if t then c1
```

where t is the test and c1 and c2 are the consequences. The test is usually a relational expression, whose value is "1"b or "0"b (that is, true or false). Each consequence is usually a single statement.

An 'if' statement that has an 'else' clause (the first form), is interpreted as follows:

1. Evaluate the test.
2. If the test value is true, then execute the first consequence; otherwise execute the second consequence.

Consider, for example:

```
if x >= 2*b
  then z = sqrt(x-2*b);
  else call err3(x);
```

Here, the test is the relational expression ' $x > 2*b$ ', the first consequence is the assignment ' $z = \text{sqrt}(x - 2*b)$ ', and the second consequence is the statement 'call err3(x)'. This 'if' statement executes the assignment statement if the argument of the square root is nonnegative; otherwise, it calls an error routine.

An 'if' statement without an 'else' clause takes no action when the value of the test is false. Consider, for example:

```
if i < 0 then i = 0;
```

Depending on whether 'i' is negative or not, this statement sets 'i' to zero or takes no further action.

An 'if' statement without a 'then' clause is not part of the language. However, a specification that requires an omitted 'then' clause can easily be converted to one that requires an omitted 'else' clause; and the resulting uniformity is beneficial. For example, consider the specification:

```
If 'x=0' is true, then do nothing; otherwise, execute the assignment
statement 'y=1/x;'.
```

This specification can be converted to a suitable form by negating the test and exchanging the consequences. The result is:

```
If 'x^= 0' is true, then execute the assignment statement 'y=1/x;';
otherwise, do nothing.
```

The specification can now be written in PL/I, as follows:

```
if x^= 0 then y = 1/x;
```

Test in an 'if' Statement

The test in an 'if' statement can be any expression that yields a scalar bit-string value. Usually, the bit-string value is of length one, but any length is allowed. If any bit of the test value is one, then the test is true; otherwise, the test is false.

The test can specify a lengthy computation; for example:

```
if abs(x) = 0
    & sqrt(u**2+v**2)<f(r+1)
    & (sw1=5 | sw1=9)
    then call R1(z);
    else call R2(z);
```

Here the test is a complicated bit-string expression; furthermore, the test contains the reference 'f(r+1)', which (it is assumed) is a reference to a user-defined function and which can require considerable computation in itself. When the computation is complete, however, the final result is simply "1"b or "0"b.

Although the definition of the test in an 'if' statement permits any expression that yields a bit-string value, an expression that does not yield a 'bit(1) nonvarying' value should be avoided. A multiple-bit test can give unexpected results; for example, consider the following statements:

```
if B then call R1; else call R2;
if ^B then call R2; else call R1;
```

It is natural to expect these statements be equivalent to one another; however, they are equivalent only if the value of 'B' is a string of zeros only or of ones only, and that can be assured only if 'B' is 'bit(1) nonvarying'. Suppose, for example, that 'B' is declared 'bit(2)' and the assignment statement

```
B = "01"b;
```

has been executed. In the first statement, the test is true because it contains a one bit and therefore 'R1' is called. In the second statement, the value of the test is '"10"b' and so it is once again true and 'R2' is called. Thus the statements are not equivalent when 'B' is declared 'bit(2)'.

Consequences in an 'if' Statement

Each consequence in an 'if' statement must be an executable unit. The constructs that are executable units are:

- The 'do' group; that is, a 'do' statement followed by a sequence of statements followed by an 'end' statement
- The 'begin' block; that is, a 'begin' statement followed by a sequence of statements followed by an 'end' statement

- The independent statement; that is, one of the following kinds of statement:

storage management: 'allocate' and 'free'
assignment: the assignment statement
flow of control: 'if', 'goto', and the null statement
procedure invocation: 'call' and 'return'
condition handling: 'on', 'revert', and 'signal'
input/output: 'open' and 'close'
stream input/output: 'get' and 'put'
record input/output: 'read', 'write', 'delete', 'rewrite' and 'locate'

The definition just given allows the use of nearly any group, block, or statement as a consequence. Those not allowed are:

- Constructs that cause no action when encountered through sequential flow of control; for example, a 'procedure' block, a 'format' statement, or a 'declare' statement.
- Statements that are not complete in themselves but must be part of a larger construct; for example, the 'do' statement, which must be part of a 'do' group, or an 'entry' statement, which must be part of a 'procedure' block.

The broad definition of the consequence means that the use of a 'goto' statement to transfer around statements is unnecessary. For example, instead of writing:

```

if x=0 & y<2 then goto L3;
do i = 1 to n;
  A(i) = B(i)*C(i);
  Q(i) = 0;
end;
L3: ...
  
```

one should write:

```

if ^(x=0 & y<2) then
  do i = 1 to n;
    A(i) = B(i)*C(i);
    Q(i) = 0;
  end;
...
  
```

This form avoids the use of a label prefix; furthermore, its syntactic structure corresponds to the logical structure of the computation.

Two of the constructs that can be used as consequences are of particular significance. They are the noniterative 'do' group and the 'if' statement itself; and they are given special attention in the following paragraphs.

NONITERATIVE 'do' GROUP AS A CONSEQUENCE

A noniterative 'do' statement is one that has no 'while' option or index. A noniterative 'do' group begins with a noniterative 'do' statement, and its sole purpose in the language is to gather together two or more statements so that they can be treated as a single consequence in an 'if' statement. Consider, first, the following two program fragments:

```
if x=0 then z=1; call Q(alpha);

if x=0 then do; z=1; call Q(alpha); end;
```

These program fragments have different meanings. The first fragment is an 'if' statement followed by a 'call' statement; and the two statements are executed independently, one after another. The second fragment is an 'if' statement whose consequence is a noniterative 'do' group; and the entire 'do' group is executed or not depending on whether the value of 'x' is zero or not.

Another example of the use of a noniterative 'do' group is:

```
if a = b
  then do;
    alpha = F(3,a-2) - F(3,2*b);
    beta  = G(0);
    gamma = 4;
  end;
  else do;
    alpha = F(3*(a-b)+1,2*(a-1)-b) - F(3,2*b);
    beta  = G((a-b)/2);
    gamma = H(2*(a-b)+4);
  end;
```

Provided such statements are laid out on the page in a clear and uniform way, they are easy to read and understand.

'if' STATEMENT AS A CONSEQUENCE

As a case of particular interest, the consequence of an 'if' statement can, itself, be an 'if' statement; that is, 'if' statements can be nested. There is no limit to the depth of this nesting, and it is not uncommon for nesting to have three levels.

Suppose that the following case analysis is given as part of the specification of a program:

<u>Test</u>	<u>Expression for z</u>
x>0	atan(y/x)
x=0 & y>0	pi/2
x=0 & y=0	(undefined)
x=0 & y<0	-pi/2
x<0 & y>=0	atan(y/x)+pi
x<0 & y<0	atan(y/x)-pi

This table gives five different expressions for 'z', depending on the values of 'x' and 'y'; it also shows the range of values for which 'z' is not defined. The case analysis can be programmed without nesting as follows:

```
if x>0 then z = atan(y/x);
if x=0 & y>0 then z = pi/2;
if x=0 & y=0 then signal error;
if x=0 & y<0 then z = -pi/2;
if x<0 & y>=0 then z = atan(y/x)+pi;
if x<0 & y<0 then z = atan(y/x)-pi;
```

This version is readable but not efficient. There are 11 relational expressions in the program fragment, and every one of them is evaluated every time the fragment is executed.

An efficient programming of the case analysis given in the table is:

```
if x>0
  then z = atan(y/x);
else if x=0
  then if y>0
    then z = pi/2;
    else if y=0
      then signal error;
      else z = -pi/2;
  else if y>=0
    then z = atan(y/x)+pi;
    else z = atan(y/x)-pi;
```

There are five relational expressions in this version (instead of 11); and, if the cases occurred with equal frequency, an average execution of the program fragment would require the evaluation of three relational expressions (instead of 11).

Dangling 'else' Clause

The omission of an 'else' clause in an 'if' statement that is part of a nest of 'if' statements can produce confusion. Consider the following statement:

```
if x<0 then if a<0 then y=a/x; else y=0;
```

Does the 'else' clause go with the entire statement (so that it is executed when 'x<0' is false) or does it go with the nested 'if' statement (so that it is executed when 'x<0' is true and 'a<0' is false). Because the answer to this question is not obvious, 'else y=0;' is called a "dangling 'else' clause".

The rules of PL/I supply the answer: an 'else' clause is always associated with the smallest possible 'if' statement. Therefore, the 'else' clause in the example goes with the nested 'if' statement, and the correct layout for the statement is:

```
if x<0
  then if a<0
    then y=a/x;
    else y=0;
```

The problem with the dangling 'else' arises when a programmer wants the 'else' clause to be associated with the entire statement. Then either of the following can occur:

1. The programmer forgets the rules and writes:

```
if x<0
  then if a<0
    then y=a/x;
    else y=0;
```

This formatting is wrong, but, unfortunately, it looks reasonable.

2. The programmer remembers the rules and writes:

```
if x<0
  then do;
    if a<0 then y=a/x;
    end;
  else y=0;
```

This version is correct but it requires the introduction of a 'do' group.

A general solution to this problem is to use an 'else' clause with every 'if' statement in a nest of 'if' statements. When there is no action for the 'else' clause to perform, a null statement can be used as the consequence. A null statement is the single character ';' and its execution causes no action.

With this approach in mind, the example can be written to make both interpretations clear. To obtain the first interpretation, write:

```
if x<0
  then if a<0
    then y=a/x;
    else y=0;
  else;
```

To obtain the second interpretation, write:

```
if x<0
  then if a<0
    then y=a/x;
    else;
  else y=0;
```

'do' GROUP

There are three kinds of 'do' group:

```
iterative 'do' without index
iterative 'do' with index
noniterative 'do'
```

A 'do' group gathers together a sequence of statements for execution as a single unit. This gathering together is the only purpose of a noniterative 'do' group. An iterative 'do' group does more: it executes the statements that are gathered together repeatedly, and is used to program loops. Every 'do' group, iterative or not, eliminates at least one 'goto' statement from a program, and the result, in most cases, is an important contribution to the clarity of the program.

The general form of a 'do' group is:

```
do ... ;
  ss
end;
```

where ss is the body of the group. The body is a sequence of any number of statements. The notation 'do ...;' is used to indicate that, at this point in the discussion, the details of the 'do' statement are being left out; details are given later, as each kind of 'do' group is described. Although label prefixes are not shown above, the 'end' statement can be preceded by one or more label prefixes, and can be the destination of a transfer of control.

A transfer of control from a 'goto' statement that is outside an iterative 'do' group to a statement that is inside the group is not valid. The only way to enter an iterative 'do' group is by flowing into or transferring to the 'do' statement at the beginning of the group.

Iterative 'do' without Index

An iterative 'do' group without an index has the form

```
do while(t);
  ss
end;
```

where t is the test and ss the body of the 'do' group. The test is defined in the same way as the test in an 'if' statement; that is, it must yield a scalar, bit-string value. The test is true if at least one bit is one and is false otherwise. The use of a test bit-string with length other than one is not recommended. When this recommendation is followed, the value of the test is "1"b or "0"b.

The iterative 'do' group without an index executes the body of the group repeatedly while the value of the test is true. The detailed interpretation is:

1. Evaluate the test.
2. If the test value is false execution is complete. Otherwise,
3. Execute the statements of the body of the group; that is, start with the first statement of the body and continue until the 'end' statement is executed.
4. Go to Step 1.

The iterated 'do' group without index can be used to write the most primitive kind of loop: one that appears to go on forever. In practice, such loops are often required. Consider, for example, the program:

```
P: proc;
  dcl (sysin,sysprint) file;
  dcl x float;
  do while("1"b);
    get list(x);
    put skip list(sqrt(x));
  end;
end;
```

This program turns Multics into a calculator of square roots; it runs until the user interrupts it. The 'do' statement shown here is used wherever the programmer must design his own loop control rather than using one of the methods of control provided by PL/I.

The iterative 'do' group without an index is well suited to control of the computation of a mathematical approximation. Suppose a quantity, 'y(x)', must be computed for a given value of 'x'. The function 'y' is not given as a formula; instead, the following user-defined functions are available for use in the program:

```

initial_guess(x)      which gives the first approximation for y(x) in
                      terms of the given x

better_guess(x,oldy)  which gives the (i)th approximation for y(x) in
                      terms of x and the (i-1)th approximation, oldy

```

The following statements compute successive approximations to 'y' until two successive approximations differ by no more than one ten-thousandth of the value of the current approximation:

```

oldy = initial_guess(x);
newy = better_guess(x,oldy);
do while(abs(newy-oldy) > .0001e0*abs(newy));
    oldy = newy;
    newy = better_guess(x,oldy);
end;

```

When this program fragment is written without the benefit of the 'do' group, it is:

```

oldy = initial_guess(x);
newy = better_guess(x,oldy);
LOOP: if ^ (abs(newy-oldy) > .0001e0*abs(newy))
    then goto DONE;
    oldy = newy;
    newy = better_guess(x,oldy);
    goto LOOP;
DONE: ...

```

This version requires two 'goto' statements and their accompanying label prefixes. It makes details explicit but obscures the general intent of the programmer.

Iterative 'do' with Index

An iterative 'do' group with an index has the form:

```

do i = cl;
    ss
end;

```

where i is the index, cl is the control list, and ss is the body of the group. The index is usually a simple reference to a scalar, but other possibilities are described later. The control list is a sequence of controls separated by commas.

Execution of an iterative 'do' group with an index is divided into phases. The first control in the control sequence governs the first phase, the second control governs the second phase, and so on. Consider the following 'do' group:

```
do i = 1 by 2 to 5, 13;  
  A(i) = B(i);  
end;
```

In this example, the control list consists of two controls. The first phase of execution is governed by the control '1 by 2 to 5' and executes the body of the group three times. The second phase is governed by the control '13' and executes the body once. The 'do' group is equivalent to

```
A(1) = B(1);  
A(3) = B(3);  
A(5) = B(5);  
A(13) = B(13);
```

There are three kinds of controls, as follows:

```
The single-valued control  
The repeat control  
The FORTRAN control
```

Detailed descriptions of these controls follow. The controls are quite different from one another, but they have the 'while' option in common, which allows a control phase to be cut short when the test in the option is false.

SINGLE-VALUE CONTROL

The single-value control has the form:

```
e while(t)
```

where e is an expression and t is the test. The expression must yield a value suitable for assignment to the index. The test is defined as in an 'if' statement. The 'while(t)' option can be omitted.

A single-value control can be used to supply an index value for a single execution of a 'do' group. The detailed interpretation is:

1. Execute the assignment statement

```
i = e;
```

2. If the 'while' option is present, evaluate the test. If the test value is false, then exit from this phase.
3. Execute the statements of the body of the 'do' group; that is, start with the first statement and continue until the 'end' statement has been executed.
4. Exit from this phase.

The 'do' group in the following program has three controls, each a single-value control:

```
P:  proc;
    del sysprint file;
    del s char(20) var;
    do s = "red", "yellow", "blue";
        put skip list("The color is "||s||".");
    end;
end;
```

The program prints:

```
The color is red.
The color is yellow.
The color is blue.
```

'repeat' CONTROL

The 'repeat' control has the form:

```
e1 repeat e2 while(t)
```

where e1 and e2 are expressions and t is the test. The expressions must yield values suitable for assignment to the index. The test is defined as in the 'if' statement. The 'while(t)' option can be omitted.

A 'repeat' control has one expression to supply the value for the first execution of the body of the group and a second expression for subsequent executions. The detailed interpretation is:

1. If this is the first execution of this step (Step 1. in the current phase, then execute the assignment:

```
i = e1;
```

Otherwise, execute the assignment:

```
i = e2;
```

2. If the 'while' option is present, evaluate the test. If the test is false, then exit from this phase.
3. Execute the statements of the body of the 'do' group; that is, start with the first statement and continue until the 'end' statement has been executed.
4. Go to Step 1.

The following program fragment prints all of the powers of two that are between 1 and 100:

```
do i = 1 repeat 2*i while(i<100);
    put list(i);
end;
```

The 'do' group is equivalent to:

```
      i = 1;
LOOP:  if^(i<100) then goto DONE;
      put list(i);
      i = 2*i;
      end;
DONE:  ...
```

The 'repeat' control is especially useful for searching a linked list. Suppose the array that holds the list is declared as follows:

```
      dcl 01 cell(500),
           02 code char(6),
           02 customer char(30),
           02 link fixed(9);
```

Suppose that when this array is in use:

- 'cell(1)' is the first member of the list.
- If 'cell(i)' is the (k)th member of the list, then 'cell(cell.link(i))' is the (k+1)th member of the list.
- If 'cell(i)' is the last member of the list, then 'cell.link(i)' is zero.

Consider the following 'do' group:

```
      do i = 1 repeat cell.link(i) while(i ^= 0);
         if cell.code(i) = given_code then goto FOUND;
         end;
      goto NOT_FOUND;
```

These statements search the list for the first member that has a 'cell.code' that is identical to that contained in 'given'.

FORTRAN CONTROL

The FORTRAN control has one of the forms:

e1 by e2 to e3 while(t)

e1 to e3 by e2 while(t)

where e1 is the initialization, e2 is the increment, e3 is the limit, and t is the test. The expressions are subject to the following restrictions, which reflect the uses to which the expressions are put:

- The initialization must be suitable for assignment to the index.
- The increment must be a suitable operand for the addition operation.
- The limit must be a suitable operand for the '<' operator.

The test is defined as in the 'if' statement. The 'while(t)' option can be omitted. Either the 'by e2' clause or the 'to e3' clause can be omitted, but not both.

A FORTRAN control is often used to supply a sequence of index values that is an arithmetic progression; indeed, in many cases, it is used to obtain the sequence of the first n integers:

1, 2, ..., n

Although such uses are simple, the full interpretation of the control is complicated. The complexity results from the concern of the designers for efficiency; specifically, from the decision to evaluate the expressions associated with the index, the initialization, the increment, and the limit only once during a phase.

The detailed interpretation of the FORTRAN control is:

1. Prepare for execution of the loop by performing the following actions in any order:
 - a. Determine the location of the storage unit designated by i (the index of the group). (In order to do this, any expressions in i , such as subscript expressions, must be evaluated). Save the location of the storage unit, and wherever i appears in these steps, use the saved location rather than re-evaluating the expressions in i .
 - b. If the 'by e_2 ' clause is present, evaluate e_2 and save the value. Use the saved value wherever e_2 appears in these steps.
 - c. If the 'to e_3 ' clause is present, evaluate e_3 and save the value. Use the saved value wherever e_3 appears in these steps.
2. If this is the first execution of this step (Step 2) in the current phase, then execute the statement:

$i = e_1$;

Otherwise, execute one of the following:

$i = i + e_2$; (if the 'by e_2 ' clause is present)

$i = i + 1$; (if the 'by e_2 ' clause is absent)

3. If the 'while' option is present, evaluate the test. If the test is false, then exit from this phase.
4. If the 'to e_3 ' clause is present, test the value of i against e_3 as follows: If the 'by e_2 ' clause is not present and $i > e_3$, then exit from this phase. If e_2 is positive and $i > e_3$, then exit from this phase. If e_2 is negative and $i < e_3$, then exit from this phase.
5. Execute the statements of the body of the 'do' group. Execute them in the normal way; do not use any of the information saved in Step 1. Start with the first statement and continue until the 'end' statement has been executed.
6. Go to Step 2.

A simple example of a 'do' group with a FORTRAN control is:

```
do x = -90 by .25 to 90;
  put skip list(x, sind(x), cosd(x));
end;
```

This group prints the values of the sine and cosine functions from -90 degrees to +90 degrees in increments of .25. It is equivalent to the statements:

```
x = -90;
LOOP:  if x>90 then goto DONE;
       put skip list(x, sind(x), cosd(x));
       x = x+.25;
       goto LOOP;
DONE:  ...
```

A short example that illustrates the consequences of Step 1 of the interpretation is:

```
a=0;
b=2;
c=10;
k=1;
do A(k) = a by b to c;
    k = k+1;
    b = -b;
    c = c-1;
end;
```

These statements are equivalent to:

```
do A(1) = 0 by 2 to 10;
    k = k+1;
    b = -b;
    c = c-1;
end;
```

If the current values of the control parameters were used for each execution of the loop, the interpretation of this loop would be extremely complicated; to start with, it would require a knowledge of all of the elements of the array 'A'.

INDEX OF A 'do' GROUP

There are two special restrictions on the index of a 'do' group, as follows:

- The index must designate a scalar value.
- The index must not designate an 'area' value.

Aside from these restrictions, the index can be any construct that is a valid target for the assignment statements that are explicitly shown in the interpretation of the 'do' group.

The following program illustrates the use of a pseudo-variable as the index of a 'do' group:

```
P:  proc;
    dcl sysprint file;
    dcl z complex float;
    do real(z) = 0 by .25 to 1;
        do imag(z) = 0 by .25 to 1;
            if z ^= 0 then
                put skip list(z,1/z);
            end;
        end;
    end;
end;
```

This program lists 'z' and '1/z' for 24 values of the complex variable 'z'. The program shows that one 'do' group can be used within another. There is no restriction on the depth of the nesting that can be used.

Non-iterative 'do'

A noniterative 'do' group has the form:

```
do;  
  s1  
  s2  
  ...  
end;
```

where s1, s2, and so on are the body of the group. The group is executed by executing the sequence of statements s1, s2, and so on, once.

The effect of the noniterative 'do' group is to gather s1, s2, and so on into a single syntactic unit. The only use for this kind of 'do' group is as a consequence of an 'if' statement, as described earlier in this section.

'goto' STATEMENT

A 'goto' statement has the form:

```
goto ref;
```

where ref is any reference that yields a scalar 'label' value. When the statement is executed, the reference is evaluated and control transfers to the statement designated by the 'label' value. The statement to which control transfers is the destination of the transfer of control. If the destination is outside of the block that contains the 'goto' statement, then the transfer causes control to exit from that block; block exits are described later, in Section XII, "Procedure Invocation." If the program is recursive, additional rules are required to determine the destination; these are given later, again under "Procedure Invocation", and they apply only to certain advanced and unusual situations.

'goto' with a Constant Reference

When the reference in a 'goto' statement is a constant reference, the destination must be a statement in the same block or in a containing block. The reference is rather restricted by the fact that a 'label' constant must be either a scalar or a one-dimensional array.

A 'goto' statement whose reference designates an element of a 'label' array constant is called a switch. An example of the use of a switch is:

```
      goto C(i+1);  
C(1):  z = x**3 + y**3; goto DONE;  
C(2):  z = x/(y**2-x); goto DONE;  
C(5):  z = 1;  
DONE:  ...
```

Because it is an array subscript, the value of 'i+1' is converted to an integer by dropping the fractional digits. Then the switch is interpreted as follows:

<u>Trunc(i+1)</u>	<u>Action</u>
0 or less	the 'subscriptrange' condition occurs
1	the assignment labeled 'C(1)' is executed
2	the assignment labelled 'C(2)' is executed
3	(undefined)
4	(undefined)
5	the assignment labelled 'C(5)' is executed
6 or more	the 'subscriptrange' condition occurs

Observe that no condition occurs when an element is missing from within the array of labels.

'goto' with a Non-Constant Reference

The use of a variable reference or a function reference in a 'goto' statement is usually confined to advanced applications or entirely avoided.

As an example of the use of a variable in a 'goto' statement, consider the procedure:

```
IGL: proc(x,a,error) returns(float);
      decl (x,a) float;
      decl error label;
      decl t float;
      t = x**2 + a**2;
      if t<0 then goto error;
      return(log(x+t));
      end;
```

This procedure transfers to the statement designated by the variable 'error' when 't' is negative. The variable 'error' is a parameter, and its value is supplied by the reference to the procedure. An example of such a reference is:

```
alpha = (beta + IGL(gamma,2,LAB))/delta;
...
LAB: put skip list("IGL failed at alpha");
      goto EXIT;
```

'local' ATTRIBUTE

The transfer of control performed by a 'goto' statement can be either local or non-local. A local transfer is one for which both the 'goto' statement and its destination are immediately contained in the same block. A nonlocal transfer is any other transfer. A local transfer can be executed at considerably less cost than a nonlocal transfer.

The PL/I interpreter can perform a local transfer efficiently only when the transfer can be recognized as such before execution. Three cases apply:

- If the reference in the 'goto' statement is a constant reference, a local transfer can be recognized by examination of the program.
- If the reference is a variable reference, a local transfer can be recognized only if the variable name is declared with the 'local' attribute.
- If the reference is a function reference, a local transfer cannot be recognized before execution.

As an example of the use of the 'local' attribute, consider the following program fragment:

```
        decl x local label;
        ...
        if z=0 then x=LAB1; else x=LAB2;
        ...
        goto x;
LAB1:   m = alpha + beta**2; goto DONE;
LAB2:   m = alpha - beta**2; goto DONE;
DONE:   ...
```

The use of the 'local' attribute is never essential. If the 'local' attribute were omitted from the declaration of 'x', these statements would still be correct; but the execution of the 'goto x;' would cost more.

RESTRICTION ON THE DESTINATION

The destination of a 'goto' statement must be a statement that is immediately contained in an active block. If the reference in the 'goto' statement is a constant reference, this restriction requires that the destination be in some block that contains the 'goto' statement, and the compiler checks for compliance. If the reference is a variable reference or a function reference, the matter is neither so simple nor so safe.

The following program illustrates the problem:

```
P:   proc;
     decl x label;
P1:  proc;
     ...
     x = LAB;
     ...
LAB: ...
     end;
P2:  proc;
     ...
     goto x;
     ...
     end;
     call P1;
     call P2;
     end;
```

This program executes the procedure 'P1' and then the procedure 'P2'. The first procedure assigns the label value designated by 'LAB' to the label variable 'x'. At the time of assignment, the statement designated by 'LAB' is an active block. Later, the second procedure attempts to execute 'goto x;'; but at that time the block that contains the statement designated by 'LAB' is no longer active. Therefore the transfer is not valid.

There is no way to detect this invalid transfer in advance without understanding the logic of the program; therefore, the error is not detected by the compiler.

BLOCK EXECUTION

The principal purpose of a 'begin' block is to establish declarations. The recommended style of programming in Multics PL/I is to write short 'procedure' blocks. Since the necessary declarations can be established in these 'procedure' blocks, a 'begin' block is rarely needed; indeed, some programmers never use them.

Control can enter a 'begin' block only by execution of the 'begin' statement that is the first statement of the block. The statement is executed either in sequence or by transfer of control to a label prefix in the statement. When the 'begin' statement is executed, it does not cause any action directly, but the entry to the 'begin' block causes the block to be activated.

Control can exit from a 'begin' block by execution of the 'end' statement that is the last statement of the block. Alternatively, control can exit by execution of a 'goto' statement whose destination lies outside the block. In either case, exit from the block causes the block to be deactivated.

The activation and deactivation of a block are described later, in Section XII, "Procedure Invocation."

GUIDELINES FOR FLOW OF CONTROL

When the PL/I facilities for programming flow of control are used effectively, they make the logic of a program seem simple and thus allow the programmer to concentrate on the details of the operations being performed. Two factors are important in PL/I programming: avoidance of unnecessary 'goto' statements and proper use of page layout. These factors are discussed here.

Avoidance of Unnecessary 'goto' Statements

A common source of unclear programming is the use of a 'goto' statement where 'do' groups or procedure calls could be used. The avoidance of these 'goto' statements requires more work in the design of a program but reduces the work of debugging; the net result is an improvement of the program.

Some uses for the 'goto' statement are necessary in PL/I because they can be avoided only by obscure tricks. The use of a switch is sometimes essential and the use of a 'goto' statement to escape from a loop is common. But every use of a 'goto' should be considered carefully and retained only if it fills a special need. There are not many reliable rules for good programming style; but avoidance of unnecessary 'goto' statements is one such rule.

Layout Conventions

A program is arranged in an attractive layout by means of blanks, tabs, and newlines. The PL/I interpreter ignores the layout of a program, and leaves this responsibility entirely in the hands of the programmer. The programmer therefore has two tasks:

- He must choose conventions that provide for program layouts.
- He must detect his own errors in his application of the layout conventions.

There is some variation in the layout conventions used by PL/I programmers. The following rules are used for the example programs in this manual:

1. Start each statement on a new line. This keeps statements from getting lost.
2. If a statement, group, or block requires more than one line, indent every additional line relative to the first line. This makes it easy to find the end of the statement, group, or block.
3. If an 'if' statement with an 'else' clause requires more than one line, begin a new line for the 'then' clause and a new line for the 'else' clause. Start the 'else' clause in the same column as the 'then' clause. This makes it easy to match the clauses of an 'if' statement.
4. Put every label prefix at the left margin, even when the statement of which it is a part is indented. This makes it easy to search for a particular label prefix.

These rules have exceptions. For example, a consequence should begin on the same line as the 'then' or 'else' that precedes it. If a statement is very closely related to the statement that precedes it, it can appear on the same line; this is sometimes true of a 'goto' statement.

SECTION XII

PROCEDURE INVOCATION

Programs can be written quickly and accurately if the top-down structured approach to programming is used. This approach is applied to a given problem as follows:

1. Call the problem the current task.
2. Program the current task as a procedure. If the task is large and complicated, factor out a subtask; that is, select a coherent portion of the task, give it a name, and replace it by a 'call' statement or function reference to a procedure that will be written later. Place the factored subtask on the list of remaining tasks. Continue to factor out subtasks in this way until the current task can be written as a simple procedure that is about one page long.
3. If there are any remaining tasks, let one of them be the current task and go to Step 2. Otherwise, the program for the given problem is complete.

The availability of a complete and efficient facility for writing and invoked procedures is useful for top-down structured programming. Although the large number of procedure invocations introduced by the approach can increase the expense of executing a program, that expense is compensated for by the reduction in the cost of program development and maintenance. Further, the Multics implementation of PL/I includes special optimizations designed to reduce the cost of procedure invocation. From all of this, it follows that procedure invocation is perhaps the most important feature of Multics PL/I.

The subject of procedure invocation is presented here in an order that allows explanation and motivation of each topic. The discussion begins with the passing of arguments to parameters since all the features of procedure invocation depend on this operation. After that preliminary, the execution of 'call' statements and the interpretation of function references are described. This provides a basis for the consideration of the special properties of procedures, including recursion and side effects. Next the use of variables and function references to supply the entry point of the invoked procedure is explained. Finally, the details of the syntax of a procedure are given.

ARGUMENTS AND PARAMETERS

Whether a procedure is invoked by a 'call' statement or a function reference, an important part of the operation is the passing of arguments to parameters. As a simple example, consider the following program:

```
P:  proc;
    dcl a float;
    dcl b fixed;
    ...
    call Q1(a,b,a+6);
    ...
Q1:  proc(x,y,z);
    dcl (x,y,z) float;
    ...
    end;
end;
```

In this example, the 'call' statement invokes the procedure labeled 'Q1'. The 'call' statement has a list of three arguments:

(a,b,a+6)

and the procedure has a corresponding list of parameters, one for each argument:

(x,y,z)

When the procedure 'Q1' is invoked, each argument is passed to its corresponding parameter.

The arguments are passed in two ways, as follows:

- If the argument is a variable reference and has a suitable storage type, then the variable designated by the argument is passed; that is, the corresponding parameter is set to designate the same variable as the argument just before procedure execution. Such an argument is a by-reference argument. The first argument in the program just given is a by-reference variable.
- If the argument is suitable for assignment to the corresponding parameter but cannot be handled as a by-reference argument, then the value designated by the argument is passed; that is, a system temporary is allocated, the argument value is assigned to the temporary, and the parameter is set to designate the temporary. Such an argument is a by-value argument. The second argument in the program is by-value because its storage type differs from that of its parameter. The third argument is by-value because it is not a variable reference.

Observe that an argument that is passed by-reference can have its value changed during procedure execution, whereas a by-value argument merely supplies the initial value for a system temporary.

The discussion of arguments and parameters given thus far is intended to be an introduction to the subject. In the following paragraphs, detailed rules and examples are given. First the classification of arguments as by-reference or by-value is defined; then the interpretation of the two kinds of arguments is given. The discussion continues with guidelines for argument usage and concludes with a note on argument validity.

Argument Classification

The rules for the classification of an argument as by-reference or by-value are given here. The classification is performed during program compilation. The ability to pass arguments by-reference can contribute to the efficiency of the object program. The existence of two kinds of arguments makes the rules more complicated; specifically, it accounts for the presence of Rules 3 and 4.

An argument is by-reference if it satisfies all of the following rules:

- A by-reference argument must be a variable reference. This restriction is present because a procedure can assign a value to a by-reference argument; and only a variable reference can be the target of an assignment statement.
- A by-reference argument must have the same storage type as the corresponding parameter. This restriction is present because a by-reference argument and a parameter designate the same variable and (with exceptions that are not relevant here) every reference to a given variable must use the same storage type.
- With one exception, a by-reference argument must be declared with constant extents. This restriction is present because argument classification is done at compile time, when the value of a nonconstant extent is not known.

The exception to this restriction allows the use of a nonconstant extent in the declaration of an argument if the corresponding extent in the declaration of the parameter is '*'. The '*' parameter extent means that the extent is copied from the argument extent; therefore it necessarily has the same value and there is no need to examine the value of the argument extent.

- A by-reference argument must not be a reference to an array that is declared as 'defined' and uses isub's in its definition. This restriction is present because a reference to an isub-defined array requires a special encodement, the need for which cannot be detected at compile time.

An argument is by-value if it does not satisfy some of the rules just given and does satisfy the following rule:

- A by-value argument must have a storage type such that the argument value can be converted, where necessary, to the storage type of the corresponding parameter. This restriction is present because the value of a by-value argument is assigned to a temporary that has the storage type of the parameter.

With one exception, an argument is valid if it can be classified as by-reference or by-value. The exception is:

- A by-reference argument must not designate an unconnected aggregate if the array bounds of the declaration of the corresponding parameter are all constants. This restriction is present because of efficiency considerations and the compile-time classification of arguments.

The term unconnected is defined in terms of the layout of storage described under "Arrays" in Section III, "Value Storage." An array is connected or unconnected, depending on whether its elements are adjacent in storage or not.

EXAMPLES OF ARGUMENT CLASSIFICATION

As the basis for detailed examples of argument classification, consider the following program:

```
P:  proc;
    dcl 01 S(10) static external,
        02 acct decimal(8,2),
        02 err entry(float,fixed dec(10)) returns(float);
    dcl B char(n) controlled;
    dcl n fixed;
    dcl 01 T,
        02 acct dec fixed(8,3),
        02 err entry(float,fixed dec(10)) returns(float);
    dcl C char(30);
    ...
    n = 30;
    allocate B;
    call Q2(S(3),B);
    ...
    call Q2(T,C);
    ...
Q2:  proc(m1,m2);
    dcl 01 m1,
        02 balance dec fixed(8,2),
        02 erout entry;
    dcl m2 char(30);
    ...
    end;
end;
```

This example has two 'call' statements. The arguments in the first 'call' statement are classified as follows:

- The argument 'S(3)' is by-reference. The declaration of 'S' is considerably different from that of 'm1', but a careful examination shows that the declarations give the same storage type: 'S' is an array of structures, but 'S(3)' is a structure of the same aggregate type as 'm1'; the attributes 'static external' are not part of the storage type; 'decimal(8,2)' and 'fixed dec(8,2)' are two ways to describe the same data type; and the only part of the declaration of 'S.err' that is part of the storage type is the keyword 'entry'.
- The argument 'B' is by-value. Although the storage types of 'B' and 'm2' are the same when the 'call' statement is executed, Classification Rule 3 is not satisfied.

The arguments in the second 'call' statement are classified as follows:

- The argument 'T' is by-value. The storage type of 'T' differs from that of 'm1' in just one place; the scale-factor of 'T.acct' is different from that of 'm1.balance'.
- The argument 'C' is by-reference. Its storage type is explicitly the same as that of 'm2'.

This example illustrates the fact that a parameter can correspond to a "by-reference" argument in one 'call' statement and to a "by-value" argument in another.

EXAMPLES OF CONNECTED AND UNCONNECTED ARGUMENTS

As the basis for examples of arguments that are connected or unconnected arrays, consider the following program:

```
P:  proc;
    decl A(12,12) float;
    ...
    call Q5(A(3,*));
    ...
Q5:  proc(X);
    decl X(12) float;
    ...
    end;
end;
```

The argument 'A(3,*)' is valid for the following reasons:

- It is classified as by-reference.
- It is a cross-section of an array but is nevertheless connected because 'A(3,1)', 'A(3,2)', and so on, are adjacent in storage. Therefore it is a valid by-reference argument for a parameter that has a constant array bound.

Suppose the 'call' statement in the example just given is changed to:

```
call Q5(A(*,8));
```

This 'call' statement is not a valid invocation of the procedure. The argument is not connected because it represents 'A(1,8)', 'A(2,8)', and so on, and the designated variables are not adjacent in storage. There are two ways to correct the call:

- The argument can be parenthesized, thus:

```
call Q5((A(*,8)));
```

For purposes of classification, the argument is now a parenthesized expression, not a variable reference. Therefore the argument is by-value and this restriction does not apply. The method cannot be used if the procedure 'Q5' is programmed to assign a value to 'A(*,8)'; and it incurs the added expense of copying the array into a system temporary as part of the passing of the argument.

- The array bound in the parameter can be changed to '*', thus:

```
decl X(*) float;
```

The parameter no longer has a constant array bound; therefore, this restriction does not apply. The argument remains by-reference, but it is handled by the more general and more expensive method associated with an '*' array bound.

The choice between these two methods does not arise often because variables that designate unconnected storage are unusual in typical programming applications.

Argument Interpretation

When a 'call' statement or function reference is executed, each argument is interpreted according to its classification, as follows:

- If the argument is by-reference, then the corresponding parameter is set, just before each procedure execution, to designate the variable that is currently designated by the argument. The parameter designates that variable throughout the execution of the procedure. Just after procedure execution the association between the parameter and the variable is broken; but the variable itself continues to exist.
- If the argument is by-value, then the interpreter allocates a system temporary that has the same storage type as the parameter, evaluates the argument, assigns the argument value to the temporary, and sets the parameter to designate the temporary. These actions are all taken just before procedure execution. The parameter designates the temporary throughout the execution of the procedure. Just after procedure execution, the interpreter frees the temporary and its value is lost.

The differences in the two interpretations lie entirely in the 'call' statement or function reference that invokes the procedure, and not in the interpretation of the procedure itself.

EXAMPLES OF ARGUMENT INTERPRETATION

As the basis for examples of the interpretation of arguments, consider the following program:

```
P: proc;
    dcl alpha(10) float;
    dcl beta float bin(40);
    dcl (i,k) fixed;
    ...
    call Q3(alpha(i+2),beta,2*k+6):
    ...
Q3:  proc(d,x1,x2);
      dcl (d,x1,x2) float;
      ...
      end;
    end;
```

The arguments in this example are interpreted as follows:

- The argument 'alpha(i+2)' is classified as by-reference. Just before procedure execution begins, the subscript expression is evaluated; suppose the current value of the expression is '3'. Then the interpreter locates the variable designated by 'alpha(3)' and sets the parameter 'd' to designate that variable. During procedure execution, the third element of 'alpha' can be referenced either as 'alpha(3)' or 'd'. It may happen that the value of 'i' is changed during procedure execution; but that change does not cause 'd' to designate some other element of the array 'alpha'. Just after procedure execution, the association between 'd' and 'alpha(3)' is broken and 'd' becomes undefined until the next time the procedure is invoked.

- The argument 'beta' is classified as by-value because its precision is different from that of the parameter 'x1'. Just before procedure execution, the interpreter allocates a system temporary of storage type 'float', evaluates the variable reference 'beta', and assigns the value to the system temporary. During procedure execution, the parameter 'd' designates the temporary, not the variable that is designated by 'beta'. If the value of 'beta' is changed during procedure execution, it does not affect the value of the temporary. Just after procedure execution, the temporary is freed.
- The argument '2*k+6' is classified as by-value. It is treated in much the same way as 'beta'; however, for '2*k+6' it is more readily apparent that a change in the temporary does not change the argument, since a value cannot be assigned to an operator expression.

Effect of an '*' Extent

When an asterisk, '*', is written as an extent in the declaration of a parameter, it indicates that the extent for the parameter is to be copied from the storage type of the corresponding argument. (An extent is an array bound, a maximum string length, or an area size.) The value of the parameter extent is determined in this way each time the procedure is invoked, and therefore can change from one invocation to another. Under certain circumstances, when the aggregate type of the argument must be converted, there is no argument extent that corresponds to a '*' parameter array bound; then, the value '1' is assumed for the array bound.

EXAMPLE OF '*' EXTENT

As an example of the use of an '*' as an array bound, consider the following program:

```
P:  proc;
    decl A(s) float controlled;
    decl s fixed;
    ...
    allocate A;
    call Q4(A);
    ...
    call Q4(200);
    ...
Q4:  proc(X);
    decl X(*) float;
    ...
    end;
end;
```

This program has two 'call' statements, and their arguments are passed as follows:

- In the first call, the argument is by-reference. The details of the program are not shown, but suppose this 'call' statement is executed twice, with 's' equal to 10 and 12, respectively. The first time, 'A' is allocated with ten elements and the declaration of the parameter becomes 'X(10) float'. The second time 'A' has twelve elements and the declaration is 'X(12) float'.

- In the second call, the argument is by-value. The argument must be converted from a 'fixed dec(3)' scalar to a 'float' array. Since the array bound is given neither by the argument nor the parameter, it is assumed to be one. Therefore, the declaration of the parameter becomes 'X(1) float'.

Guidelines for Arguments

The classification of arguments is performed automatically by the compiler, without an explicit indication in the program. This convenient arrangement must be supervised carefully by the programmer, and he should be aware of the classification of each argument as he writes a 'call' statement. The effect of classification on both efficiency and correctness of a program is considered here.

First consider efficiency. The cost of passing an argument becomes significant when the storage type of the argument has one or more extents; such an argument designates a relatively complicated and potentially large variable. If the argument is by-reference, it is necessary only to set the parameter to designate the argument variable; this is an inexpensive operation, whose cost is independent of the extents of the variable. On the other hand, if the argument is by-value, a temporary must be allocated and the entire value copied into the temporary; and this can be expensive. A purpose of the '*' parameter extent is to allow the passing of arguments by-reference that would otherwise be passed by-value because their extents did not agree with those chosen for the parameter.

Next consider correctness. A serious mistake is the assumption that an argument is by-reference when it is not; the assumption is easily made when the argument is a variable reference. Suppose the procedure 'Q4' in the most recently given example delivers a value by assigning it to the parameter, 'X'. This is correct because the argument 'A' is by-reference and will receive the delivered value. Suppose, however, that the programmer decides, at the last minute, to increase the precision of 'A' and therefore changes its declaration to:

```
dcl A(s) float(40) controlled;
```

Now the storage type of the argument is different from the parameter and the argument is classified as by-value. Instead of changing the value of 'A', the procedure changes the value of a system temporary, and the delivered value is lost. This mistake is serious because it is not detected by the compiler and is not apparent to a casual reader.

'call' STATEMENT

A 'call' statement has one of the following forms:

```
call ref( arglist );
```

```
call ref();
```

```
call ref;
```

where ref is the entry reference and arglist is the argument list. The entry reference must yield a scalar entry value. The argument list is a sequence of arguments separated by commas, and each argument is an expression. As special cases, the argument list can be empty, as shown in the second form, or the parenthesized argument list can be absent, as shown in the last form; these special cases are equivalent and are used when no arguments are required.

Agreement of Call with Entry Point

The entry reference of a 'call' statement designates a procedure entry point. The form of the 'call' statement must agree with the form of the procedure entry point. Specifically, the 'call' statement must satisfy the following restrictions:

1. The number of arguments in the 'call' statement must be equal to the number of parameters at the designated procedure entry point. This restriction provides for the matching of arguments with parameters. If there are no arguments in the 'call' statement, then there must be no parameters at the designated procedure entry point.
2. Each argument must be a valid by-reference argument or a valid by-value argument as defined earlier in the discussion of classification under "Arguments and Parameters".
3. A 'returns' attribute must not appear at the designated procedure point. This restriction is present because a 'returns' attribute only makes sense if the procedure is invoked by a function reference.

Execution of 'call' Statements

The execution of a 'call' statement is performed in five steps, as follows:

```
Interpret entry reference  
Interpret arguments  
Activate procedure  
Execute procedure  
Exit from procedure
```

These steps are now considered in detail.

INTERPRET ENTRY REFERENCE

The first step in the execution of a 'call' statement is the interpretation of the entry reference, as follows:

1. The entry reference is evaluated to yield an entry value.
2. The entry value is used to locate a procedure entry point. In a PL/I procedure, the procedure entry point is either a 'procedure' statement or an 'entry' statement. The procedure that immediately contains the designated procedure entry point is the invoked procedure for this execution of the 'call' statement.

In the simplest case, the invoked procedure is part of the current PL/I program. In other cases, it is a procedure that is written in some other programming system or it is a program that is written in PL/I but that is part of the Multics system, such as a subroutine described in the MPM. In all cases, it is useful to think of the procedure as being written in PL/I and to rely upon the documentation of the procedure to specify the ways in which the procedure differs from a procedure that is part of the current PL/I program.

INTERPRET ARGUMENTS

The next step in the execution of a 'call' statement is the interpretation of arguments, which was described in detail under "Arguments and Parameters". Briefly:

- Each by-reference argument is interpreted by evaluating any subscript expressions or locator-qualifier expressions that appear in the argument. Then the designated variable is located and the parameter that corresponds to the argument is set to designate that variable.
- Each by-value argument is evaluated. Then a system temporary is allocated that has the same storage type as the corresponding parameter, the value of the argument is assigned to the temporary, and the parameter is set to designate the temporary.

The order in which the arguments are interpreted is not defined. After this step, each parameter designates either a variable or a temporary.

ACTIVATE PROCEDURE

The third step is the activation of the procedure. A new activation region is created for the procedure (in the language of Multics, a new stack frame is pushed onto the stack). Storage units are allocated in this activation region as follows:

- The variables of storage class 'automatic' declared in the invoked procedure are allocated. Since the 'automatic' is assumed when no storage class attribute is given, most variables of a typical program are handled in this way.

- A system temporary for the return address is allocated. The return address is used to resume execution after the 'call' statement when execution of the invoked procedure is complete.
- Other system temporaries necessary for maintaining the stack, handling 'on' conditions, holding temporary results, and so on are allocated. These need not be considered in detail since the programmer does not refer to them directly.

In order to reduce execution cost, the Multics implementation of PL/I avoids the creation of a block activation region for certain procedures. This optimization technique does not change the interpretation of a program, and it can be ignored by a programmer who remains within the framework of the PL/I language. However, when the programmer uses various Multics routines to examine stack frames directly, the effects of the technique are apparent. The compiler prints a message for each procedure that is compiled without an activation region, and provides information about the distribution of the storage units that would have occupied the missing activation region.

EXECUTE PROCEDURE

The fourth step is the execution of the procedure. Execution begins with the 'procedure' or 'entry' statement that is at the procedure entry point. The execution of this statement causes no action because its only purpose is to describe a procedure entry point; but it provides the starting point for sequential execution of statement of the procedure.

Execution of the procedure continues until an exit statement is executed, as described in the next step.

EXIT FROM PROCEDURE

The last step in the execution of a 'call' statement is the execution of an exit statement. Each exit statement specifies an exit destination; that is, the statement that is executed once the execution of the 'call' statement is complete. There are three kinds of exit statement, as follows:

- A 'return' statement is an exit statement for the given procedure if it is contained in that procedure but not in a smaller procedure. Its exit destination is the statement that appears next after the given 'call' statement.
- An 'end' statement is an exit statement for the given procedure if it is the last statement of that procedure. Its exit destination is also the statement that appears next after the given 'call' statement.
- A 'goto' statement is an exit statement for the given procedure if it is part of the given procedure or of a more recently invoked procedure and its destination lies outside the given procedure. The exit destination is the destination of the 'goto' statement.

If the exit statement is 'return' statement, it must not have a returned result; that is, it must have the form:

```
return;
```

This restriction reflects the fact that a 'call' statement does not expect a returned value.

As part of the execution of the exit statement, the following operations are performed:

- The invoked procedure is deactivated; that is, the variables and temporaries that are allocated in the activation region for the procedure are freed and the activation region is discarded. (In the language of Multics, the stack frame associated with the procedure is popped from the stack.)
- Each system temporary allocated for a by-value argument is freed.

After these operations, control is transferred to the statement designated by the exit destination.

Examples of 'call' Statements

As an example of the execution of a 'call' statement, consider the following program:

```
P:  proc;
    dcl alpha float;
    ...
    call RA(alpha+1);
    ...
LAB: ...
    ...
RA:  proc(x);
    dcl x float;
    ...
    call RB(x);
    ...
    end;
RB:  proc(y);
    dcl y float;
    dcl A(20) float;
    ...
    goto LAB;
    ...
    end;
end;
```

The external procedure 'P' is invoked from Multics, which executes the command 'P', which is equivalent to the following PL/I statement:

```
call P;
```

Execution of the command proceeds as follows:

1. Interpret Entry Reference to P. The 'P' in the 'call' statement designates the first statement of the program.
2. Interpret Arguments for P. There are no arguments.
3. Activate P. An activation region for the procedure named 'P' is pushed onto the stack, allocating the 'automatic' variable 'alpha' and the required temporaries.
4. Execute P. Execution begins at the designated 'procedure' statement and proceeds until the first 'call' statement is reached. The execution of the 'call' statement proceeds as follows:
 - a. Interpret Entry Reference to RA. The 'RA' in the 'call' statement designates the second 'procedure' statement in the program.
 - b. Interpret Arguments for RA. The argument 'alpha+1' is by-value. A system temporary of storage type 'float' is allocated, the value of the argument is assigned to it, and the parameter 'x' is set to designate the temporary.
 - c. Activate RA. An activation region for the procedure named 'RA' is created and the required system temporaries are allocated in it.
 - d. Execute RA. Execution begins with the 'procedure' statement labeled 'RA' and proceeds until the second 'call' statement of the program is reached. The execution of the 'call' statement proceeds as follows:
 - 1) Interpret Entry Reference to RB. The 'RB' in the 'call' statement designates the third procedure statement in the program.
 - 2) Interpret Arguments for RB. The argument 'x' is by-reference. The parameter 'y' is set to designate the storage unit designated by 'x', which is the system temporary created in Step 2, above.
 - 3) Activate RB. An activation region for the procedure named 'RB' is created, and the array 'A' and the system temporaries are allocated in it.
 - 4) Execute RB. Execution begins with the 'procedure' statement labeled 'RB' and proceeds until the 'goto' statement is reached. This statement is the exit statement for 'RB' because its destination is outside of 'RB'. It is also the exit statement for 'RA' because, even though it is not contained in 'RA' it is contained in a more recently invoked procedure.
 - 5) Exit from RB. The procedure 'RB' is deactivated by freeing 'A' and its temporaries and discarding its activation region. Its argument is by-reference, so there is no argument temporary to free. This concludes execution of the statement 'call RB(x);'.
 - e. Exit from RA. The procedure 'RA' is deactivated by freeing its temporaries and discarding its activation region. The storage temporary used for its argument is freed. This concludes execution of the statement 'call RA(alpha+1);'.

5. Continue Execution of P. Execution of the external procedure continues, starting with the statement labeled 'LAB' and proceeds to the 'end' statement that is the last statement of the program is reached. This statement is the exit statement for 'P'.
6. Exit from P. The procedure 'P' is deactivated by freeing 'alpha' and its temporaries and discarding the activation region. This concludes execution of the program.

The most important feature of this example is the fact that the 'goto' statement serves as the exit statement for both the procedure named 'RB' and the procedure named 'RA'.

FUNCTION REFERENCES

A function reference has one of the following forms:

ref(arglist)

ref()

where ref is the entry reference and arglist is the argument list. The entry reference must yield a scalar entry value. The argument list is a sequence of arguments separated by commas, and each argument is an expression. The second form is used when no arguments are required.

Agreement of Reference with Entry Point

The entry reference of a function reference designates a procedure entry point. The form of the function reference must agree with the form of the procedure entry point as follows:

1. The number of arguments in the function reference must be equal to the number of parameters at the designated procedure entry point.
2. Each argument must be a valid by-reference argument or a valid by-value argument as defined earlier in the discussion of classification and validity under "Arguments and Parameters".
3. A 'returns' attribute must appear at the designated procedure entry point. The attribute is necessary because it specifies the storage type of the result returned by the function and thus specifies the storage type of the function reference itself.

When a procedure is invoked by the interpretation of a function reference, it is sometimes called a function; however, that terminology is not used here.

Interpretation of Function References

The interpretation of a function reference has six steps, as follows:

- Interpret entry reference
- Interpret arguments
- Activate procedure
- Execute procedure
- Exit from procedure
- Fetch result

The first four steps are the same as for the execution of a 'call' statement. The fifth step is different and the sixth step is new. These last two steps are now considered in detail.

EXIT FROM PROCEDURE

The fifth step in the interpretation of a function reference is the execution of an exit statement. There are only two kinds of exit statement for a function reference, as follows:

- A 'return' statement is an exit statement for the given procedure if it is contained in that procedure but not in a smaller procedure. Its exit destination is that point in the program at which the value of the function reference is about to be used.
- A 'goto' statement is an exit statement for the given procedure if it is part of the given procedure or of a more recently invoked procedure and its destination lies outside the given procedure. The exit destination is the destination of the 'goto' statement. In this case, the procedure does not return a value.

If the exit statement is a 'return' statement, it must have a returned result; that is, it must have the form:

```
return(r);
```

where r is an expression. The 'end' statement that is the last statement of the given procedure is not a valid exit statement and must not be executed. Both of these restrictions reflect the fact that a procedure that is invoked by a function reference must return a value unless it exits with a 'goto' statement.

As part of the execution of the exit statement, the following operations are performed:

- The invoked procedure is deactivated; that is, the variables and system temporaries that were allocated in the activation region for the procedure are freed and the activation region is discarded.
- Each system temporary that was allocated for a by-value argument is freed.

- The result is returned. The expression in the 'return' statement is evaluated, and the value is assigned to the storage allocated by the caller for the returned result. More is said of the way a result is returned later in this section under "The 'returns' Attribute".

After these operations, control is transferred to the point designated by the exit destination.

FETCH RESULT

As the last step in the interpretation of a function reference, the result is fetched from the temporary to which it was assigned by the preceding step. This value is the value of the function reference.

Example of a Function Reference

As an example of the interpretation of a function reference, consider the following program:

```
P: proc;
    dcl (alpha,beta) float;
    ...
    alpha = 2*F1(beta);
    ...
F1:  proc(x) returns(float);
    dcl x float;
    dcl y fixed;
    ...
    return(y+1);
    ...
    end;
end;
```

The execution of the assignment statement begins with the interpretation of the function reference, as follows:

1. Interpret Entry Reference. The 'F1' in the function reference designates the second 'procedure' statement in the program.
2. Interpret arguments. The argument 'beta' is by-reference. The parameter 'x' is set to designate the storage unit designated by 'beta'.
3. Activate Procedure. An activation region for the procedure named 'F1' is pushed onto the stack, allocating the 'automatic' variable 'y' and all required temporaries.
4. Execute Procedure. Execution begins with the 'procedure' statement labeled 'F1' and proceeds until the 'return' statement is reached.
5. Exit from Procedure. The expression 'y+1' is evaluated; then the resulting value is converted to 'float' and returned to the caller. The procedure 'F1' is deactivated by popping its activation region from the stack.

After the function reference is interpreted its result is multiplied by two and assigned to 'alpha'.

PROCEDURES

A procedure gathers a sequence of statements together for execution as if they were a single operation. A procedure is defined as:

- A 'procedure' statement, followed by
- A sequence of statements that is the body of the procedure and that may include some 'entry' statements and 'return' statements, followed by
- An 'end' statement.

Procedures, 'begin' blocks, and 'do' groups must be nested with respect to one another, as described earlier, in Section V, "Program Syntax."

Many examples of procedures are given throughout this section, and therefore no additional examples are given here. Instead, the specialized statements used in procedures are described, beginning with the 'procedure' statement, continuing with the 'entry' statement, and concluding with the 'return' and 'end' statements.

'procedure' Statement

A 'procedure' statement has one of the following forms:

```
px procedure( parlist ) [attribute ...] recursive ;
px procedure( ) [attribute ...] recursive ;
px procedure [attribute ...] recursive ;
```

where px is the prefix, 'procedure' can be abbreviated as 'proc', parlist is the parameter list, and the attribute may be the 'returns' attribute, either the 'reducible' or 'irreducible' attribute, or the 'option' attribute. The brackets indicate that the 'returns' attribute and the 'recursive' keyword are optional. The second and third forms are equivalent and are used for an entry point with no parameters.

The execution of a 'procedure' statement causes no action; its purpose is to indicate the beginning of a procedure and to define an entry point to a procedure. Descriptions of the prefix, the parameter list, the 'returns' attribute, and the 'recursive' keyword follows.

The prefix of a 'procedure' statement is a sequence of any number of condition prefixes followed by a sequence of one or more label prefixes. Each condition prefix applies to the entire procedure that begins with the 'procedure' statement; the effects of these condition prefixes are described later, in Section XIII, "Condition Handling." Each label prefix is an identifier followed by a colon, and its effect is described here.

An identifier in a label prefix in a 'procedure' statement is a procedure name, and it is interpreted as follows:

- A procedure name has the attributes 'internal entry constant' or 'external entry constant', depending on whether it is at the beginning of a procedure that is contained in a larger procedure or not.
- A 'procedure' statement begins a given procedure, but each label prefix that begins the 'procedure' statement is considered to be outside the procedure. Thus, the declaration of an 'internal' procedure name is established in the procedure or 'begin' block that immediately contains the given procedure; and the declaration of an 'external' procedure name is established in an imaginary 'begin' block that encloses all of the external procedures of a program.
- During execution of the program, a reference to the procedure name yields the 'entry' value that designates the 'procedure' statement.

The prefix of a 'procedure' statement usually consists of just a single label prefix, as in the following example:

```
P3:  proc(x);
```

During the informal discussion of programs, it is convenient to use the identifier in a label prefix both as a name for a procedure and as a name for an entry point of the procedure. For example, it might be said of the statement 'call P3(a+1);' that it "invokes the procedure 'P3'" or that it "invokes a procedure at entry point 'P3'". Strictly speaking, however, the statement "invokes a procedure at the entry point that is designated by the 'entry' value that is associated with the name 'P3'".

The following 'procedure' statement has a prefix that consists of three label prefixes:

```
ALPHA:
Top_run:
F13:   proc(y);
```

The three 'entry' constant names, 'ALPHA', 'Top_run', and 'F13', all designate the same entry point. The layout used in this example is recommended because it places each name at the left margin and makes it easy for a reader to search for an entry name.

PARAMETER LIST

The parameter list is a sequence of parameters separated by commas, and each parameter is an identifier. Parameters should be declared; and, for clarity, the declarations of the parameters should be the first statements of the procedure body.

```
AA:  proc(alpha,Max_Val);
      dcl alpha fixed dec(8);
      dcl Max_Val float;
      ...
      end;
```

With the following exceptions, a parameter name is declared like any other variable name:

- When an identifier appears in a parameter list, its scope, storage class, and category must be 'internal', 'parameter', and 'variable', respectively. Since these attributes are supplied by default, there is no need to write them in the declaration of a parameter.
- The 'initial' attribute cannot be used. This exception is present because the 'initial' attribute is valid only for a variable that has its own storage, and a parameter is always set to designate previously allocated storage.
- An extent (array bound, maximum string length, or area size) can be written only as an '*' or as a decimal integer constant. The use of the '*' extent is discussed earlier, under "Arguments and Parameter"; it indicates that the extent for the parameter is to be copied from the storage type of the corresponding argument. A decimal integer constant does not provide for anything that a '*' cannot do, but it permits more efficient compilation of references to the parameter, and should be used where the generality of an '*' extent is not required.

These exceptions reflect the logical consequences of the way in which parameters are interpreted; they do not restrict the parameters. Indeed, an important feature of PL/I is that any value, scalar or aggregate, computational or noncomputational, can be passed as an argument to a procedure or returned as a result.

'returns' ATTRIBUTE

Depending on whether the 'returns' attribute is present or absent, an entry point is invoked by a function reference or a 'call' statement. The 'returns' attribute has the following form:

```
returns( d )
```

where d is the descriptor. The descriptor gives the declaration of the storage type of the returned result; with the following exceptions, it is written exactly as the storage type of a variable is given in a 'declare' statement:

- Any attributes that are not part of the storage type are omitted.
- The names of members of a structure are omitted.
- Each extent (array bound, maximum string length, or area size) must be either a '*' or a decimal integer constant.

The storage type given by the descriptor of the 'returns' attribute describes the system temporary to which the 'return' statement assigns its value. This temporary is associated with the function reference that invoked the procedure.

Example of the 'returns' Attribute

For a rather complicated example of a descriptor, consider the following procedure:

```
A1:  proc(X) returns(01 dim(*),
                    02 float,
                    02 fixed dec(8));
      decl 01 X(*),
           02 num float,
           02 m fixed dec(8);
      return(X+1);
      end;
```

This procedure accepts an array of structures with any bounds as its parameter and it returns a result that has exactly the same storage type. It is chosen as an example because the declaration of 'X', given in a 'declare' statement, can be compared to the declaration of the result, given as a descriptor in the 'returns' attribute.

'recursive' KEYWORD

The 'recursive' keyword has the form:

```
recursive
```

This keyword applies to an entire procedure, not to a particular entry point; therefore, it is used only in a 'procedure' statement and is not called an attribute. Furthermore, the keyword refers to the way in which a procedure is invoked and not to the action taken by the procedure.

A procedure is recursive if, for some possible execution of the program in which it appears, it is invoked when a previous invocation is still active; examples are given later, under "Recursive Procedure Execution". In Standard PL/I, every recursive procedure must have the keyword 'recursive' in its 'procedure' statement. In some implementations of PL/I, the addition of the keyword causes a change in the way the procedure is compiled; specifically the code becomes valid for recursion but less efficient. The Multics implementation of PL/I assumes that every procedure is recursive and thus does not depend on the 'recursive' keyword. The appropriate use of the 'recursive' keyword is recommended to Multics programmers because it maintains compatibility with the Standard and gives useful information about the procedure to a reader.

'entry' Statement

An 'entry' statement has one of the forms:

```
px entry( parlist ) [returns( rd )] ;  
px entry( ) [returns( rd )] ;  
px entry [returns( rd )] ;
```

where px is the prefix, parlist is the parameter list, and rd is the result descriptor. An 'entry' statement is the same as a 'procedure' statement except for the following points:

- The 'entry' statement does not appear at the beginning of a procedure; instead, it can occur at any point in the body of the procedure. It is used when more than one entry point is required for a procedure.
- The prefix of an 'entry' statement must not contain a condition prefix.

The last exception is present because the condition prefixes are given, once and for all, in the 'procedure' statement that begins the procedure.

The execution of an 'entry' statement causes no action; its sole purpose is to define an entry point to a procedure. Thus when an 'entry' statement is encountered by sequential flow of control through the procedure, the 'entry' statement has no effect and control continues to the next statement.

EXAMPLE OF A MULTIPLE-ENTRY PROCEDURE

Two or more procedures can be usefully combined into a single procedure when they have statements or data in common. The following procedure has two entries, one for use with two arguments, and the other for use with one argument.

```
P2A: proc(x,ypar);  
    dcl (x,ypar) float;  
    dcl y float;  
    y = ypar;  
    goto START;  
P1A:  entry(x);  
      y = 1;  
START: ...  
      end;
```

This procedure has two entries. The first entry has two arguments. The second entry has only one argument and provides the value '1' for the second argument. The handling of the parameters seems awkward, but the following simpler version is not valid.

```
P2A: proc(x,y);
      dcl (x,y) float;
      goto START;
P1A:  entry(x);
      y = 1;
START: ...
      end;
```

This program is invalid because when it is invoked at its second entry, 'P1A', it uses the parameter 'y' which is not defined for that entry.

'return' Statement

The 'return' statement has one of the following forms:

```
return( re );
return;
```

where re is the return expression. The first form is a valid exit statement for a procedure that was invoked by a function reference; the second form, for a procedure that was invoked by a 'call' statement.

'end' Statement

The 'end' statement has the following form:

```
end;
```

The purposes of the 'end' statement are to indicate the end of a procedure (or a 'begin' block or a group) and to serve as an exit statement for a procedure. An 'end' statement is a valid exit statement for a procedure only if the procedure was invoked by a 'call' statement.

ENTRY REFERENCES

The entry reference of a 'call' statement or a function reference specifies the entry point for the invoked procedure. In addition, the entry reference must supply the storage type of each parameter and, if it is present, the 'returns' attribute.

An entry reference is usually a constant reference to an entry point in the same external procedure. That case is handled in a simple way in PL/I, and the examples given thus far have all used that kind of entry reference. In what follows, the constant entry reference is given further consideration and then other kinds of entry references are introduced.

Constant Entry References

An entry reference can designate an entry point in the same external procedure, in a different external procedure, or outside the program. Each case requires special consideration when the entry reference is a constant reference.

ENTRY POINTS IN THE SAME EXTERNAL PROCEDURE

When a constant entry reference designates an entry point that is within the same procedure, the compiler can obtain the information it requires from the designated 'procedure' statement or 'entry' statement. Therefore, no additional declaration is required. This consideration applies to external entry names in the same procedure as well as internal entry names.

ENTRY POINTS IN ANOTHER EXTERNAL PROCEDURE

When a constant entry reference designates an entry point that is in another external procedure of the program, the compiler cannot obtain information directly about the parameters and the returned value because external procedures are compiled separately. Therefore, the programmer must refer to the invoked entry point, obtain the information, and declare the entry name in the procedure in which it is called by means of a 'declare' statement.

The 'declare' statement for the 'entry' constant name must have the following attributes:

- An 'entry' attribute with parameter descriptors. The keyword 'entry' is followed by a parenthesized list of descriptors, one for each of the parameters at the entry point. The descriptor is obtained from the declaration of the parameter by deleting attributes that are not part of the storage type and deleting any names of members of a structure.
- The 'returns' attribute (if one appears at the invoked entry point).

When an 'entry' name is declared in this way, its scope and category are assumed to be 'external constant'.

As an example of the declaration of a constant name that designates an entry point in another external procedure, consider the following program:

```
P:  proc;
    dcl B3 entry(01 dim(*),
                02 float,
                02 float,
                fixed)
        returns(float);
    dcl (i,j) fixed;
    ...
    i = B3(0,j);
    ...
end;

B3: proc(a,b) returns(float) recursive;
    dcl 01 a(*),
        02 Q1 float,
        02 Q2 float;
    dcl b fixed;
    ...
end;
```

Because of the 'declare' statement in the first external procedure, 'P', the compiler can compile the procedure separately and yet produce the correct code for the statement 'i=B3(0,j);'.

ENTRY POINTS OUTSIDE THE PROGRAM

When a constant entry reference designates an entry point of a procedure that is outside of the current program, the entry name must be declared by a 'declare' statement. This situation arises when a program must call one of the standard Multics subroutines or a subroutine programmed in some language other than PL/I. The 'entry' attribute with parameter descriptors, the 'returns' attribute, and the 'recursive' keyword are obtained from the documentation of the procedure. The documentation may call for the 'options' attribute.

'options' Attribute

The 'options' attribute in Multics PL/I has the form:

```
options(o, ...)
```

where o is an option name. The most frequently used option name is 'variable'. The construct 'options(variable)' indicates that an entry point accepts a variable number of arguments or allows the attributes of an argument to change from one invocation to the next. The attribute never applies to a procedure written in accordance with this manual. It does apply to some of the standard Multics subroutines.

Variable Entry References

When an entry reference is a variable reference, the compiler cannot obtain information directly about parameters and 'returns' attribute because it cannot predict which entry point will be designated by the entry reference. Therefore, the programmer must include the required information in the declaration of the variable name.

The declaration of an entry variable reference must include the following attributes:

- An 'entry' attribute whose parameter descriptors agree with the parameter declarations of every entry point that will be designated by the entry reference.
- A 'returns' attribute that is identical to that given (if any) for every entry point that will be designated by the entry reference.
- The category attribute 'variable'.

The category attribute must be given because the default is 'constant' when the data type is 'entry'. The scope, storage class, and 'initial' attributes can be chosen according to considerations that would govern the declaration of any kind of variable.

During execution of the program, the evaluation of a variable entry reference must yield an 'entry' value that designates a valid entry point.

EXAMPLE OF ENTRY VARIABLE REFERENCE

As an example of the use of an entry variable reference, consider the following program:

```
P:  proc;
    decl v entry(dim(10) float, fixed)
        returns(float) static variable;
    decl vx entry variable;
    decl A(10) float;
    decl (m,p) fixed;
    ...
    if m=0 then vx = F1; else vx = F2;
    ...
    v = vx;
    ...
    p = 3*v(A,3)+1;
    ...
F1:  proc(a1,b1) returns(float) recursive;
    decl a1(10) float;
    decl b1 fixed;
    ...
    end;
F2:  proc(a2,b2) returns(float) recursive;
    decl a2(10) float;
    decl b2 fixed;
    ...
    end;
end;
```

The program uses 'entry' variables as follows:

- At the top of the program, 'v' is declared as an 'entry static variable' with an initial value. The remainder of the declaration is included not to describe the storage used for the variable but rather to allow the use of the variable as an entry reference.
- Next 'vx' is declared 'entry'. Since this variable is not used as an entry reference, parameter descriptors and 'returns' attribute are omitted from its declaration.
- At some time during the program, the 'if' statement assigns the 'entry' value designated by the constant name 'F1' or 'F2' to 'vx'.
- Later, the value of 'vx' is assigned to 'v'.
- Each time the assignment to 'p' is encountered, the variable name 'v' is used as an entry reference. The declaration of 'v' allows the compiler to determine whether the arguments are by-name or by-value and to determine the storage type of the returned result (and thus of the function reference 'v(A,3)' itself).

Function Entry References

When an entry reference is a function entry reference, the programmer must include parameter descriptors and 'returns' attribute in the 'returns' attribute of the function name.

EXAMPLE OF AN ENTRY FUNCTION REFERENCE

As an example of the use of an entry function reference, consider the following program:

```
P: proc;
    dcl A(10) float;
    dcl (m,p) fixed;
    ...
    p = 3*E(m)(A,3)+1;
    ...
E:   proc(a) returns(entry(dim(10) float,fixed)
        returns(float));
        dcl a fixed;
        if a=0 then return(F1); else return(F2);
        end;
F1:  proc(a1,b1) returns(float) recursive;
        dcl a1(10) float;
        dcl b1 fixed;
        ...
        end;
F2:  proc(a2,b2) returns(float) recursive;
        dcl a2(10) float;
        dcl b2 fixed;
        ...
        end;
end;
```

Compare this program to the example given for variable entry references. In this program, the selection between 'F1' and 'F2' still depends on 'm', but it is carried out by a function reference, 'E(m)'.

Generic Entry Names

A generic entry name is used to designate one of a set of entry points. The programmer specifies, in the declaration of the generic entry name, how a single entry point is selected for a particular use of the name. The selection is done during compilation, and it is based on the attributes of the arguments that accompany a particular use of the name.

EXAMPLE OF A GENERIC ENTRY NAME

As an example of the declaration and use of a generic entry name, consider the program:

```
P:  proc;
    dcl Q generic(QV when(varying,*),
                 QNV when(nonvarying,*));
    dcl S char(6);
    dcl T char(100) varying;
    dcl i fixed;
    ...
    call Q(S,i);
    ...
    call Q(T,20);
    ...
QV:  proc(a,b);
    dcl a char(*) varying;
    dcl b fixed;
    ...
    end;
QNV: proc(m,n);
    dcl m char(*) varying;
    dcl n fixed;
    ...
    end;
end;
```

Presumably, 'QV' and 'QNV' are two procedures that do nearly the same thing (so that it is logical to think of them as being the same procedure), but one is designed for the efficient handling of a 'varying' string and the other for a 'nonvarying' string. During the compilation of this program, the two 'call' statements are replaced by:

```
call QV(S,i);
call QNV(T,20);
```

Thus all references to 'Q' are eliminated.

The full description of the generic entry name is given in the PL/I Language Manual. The facility is not described fully here because its usefulness is limited.

RECURSIVE PROCEDURE EXECUTION

Sometimes a procedure is invoked when a previous invocation is still active. Such a procedure is recursive. Recursion can be direct or chained. Direct recursion results if the body of a given procedure contains a 'call' statement or function reference that invokes the given procedure. Chained recursion occurs when a procedure A invokes a procedure B that invokes a procedure C and so on until a procedure invokes procedure A. In either case, the keyword 'recursive' should be used in the 'procedure' statement at the beginning of the procedure.

In a software system that has a well-developed interrupt system, recursion can occur without being explicitly programmed. This kind of recursion frequently occurs in Multics. Suppose, for example, that a user starts printing a file, interrupts, and then starts printing another file. In this case, the program for printing a file is active on two levels, and is therefore recursive. As a less obvious example, suppose a user interrupts a file print-out to compile a program. Since the routine that was outputting a line for the file print-out may also be used by the compiler to output a message, recursion can occur in this case also. In both cases, the recursion is caused not by the programs themselves but rather by the user, who uses an interrupt to introduce a recursive call on a system routine. This possibility must be considered whenever a system routine is written.

When a procedure is executed recursively, the procedure has more than one activation internal region. These are the storage regions that, in the Multics implementation of PL/I, are the stack frames. An activation region is used for 'automatic' variables, for the extent values of 'defined' variables, for the pointers associated with 'parameter' variables, and for various system temporaries required for the evaluation of expressions, return from a procedure, and so on. A complete description of the role of activation regions is given earlier, in Section VII, "Storage Management."

The simultaneous existence of several activation regions for a recursive procedure raises the following question:

When a name denoting storage in an activation region is interpreted, which activation region is chosen?

For most cases of recursion, the following simple rule answers this question:

A reference to an activation region for a given block uses the most recently, created activation region that exists for that block.

The situations in which this rule may fail are described later in this section under the heading General Recursion. Before that point, several examples of recursion are given that use only the simple rule above.

EXAMPLE OF RECURSION WITHOUT ARGUMENTS

The following program uses surface recursion to print a symmetrical list of integers:

```
R1:  proc;
      dcl (sysin,sysprint) file;
      dcl n fixed;
      get list(n);
      call Seq;
Seq:  proc;
      dcl i fixed;
      i = n;
      put skip list(i);
      if n > 1 then
        do;
          n = n-1;
          call Seq;
        end;
      put skip list(i);
      end;
end;
```

When this program is executed and the input value is '3', the output is:

```
3
2
1
1
2
3
```

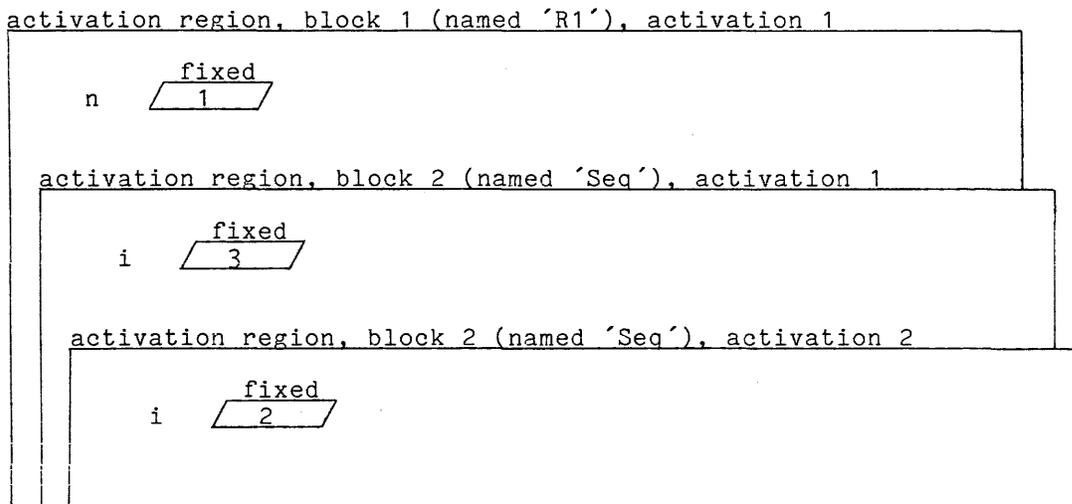
Thus the program prints the integers from the given value down to '1' and then from '1' back up to the given value.

This program uses neither statement address variables nor 'on' statements, so it is immediately recognized as a case of surface recursion. The program is short and simple and therefore it is practical to describe its execution in detail, as follows:

1. Prepare for Program Execution. The external and static regions for the entire program are created. Two 'file' constant storage units are allocated in the external region. Then 'R1' is invoked.
2. Start Execution of 'R1'. The activation region for 'R1' is created, and storage for 'n' is allocated in the region. The 'get' statement reads an input value (which is assumed to be '3') and assigns it to 'n'. The 'call' statement invokes 'Seq', which is executed as follows:
 1. Start the First Execution of 'Seq'. An activation region is created for 'Seq' and storage for the automatic variable named 'i' is allocated in it. The value of 'n' is assigned to 'i' and is printed as the first line of the output. Since 'n' is greater than one, the value of 'n' is reduced by one and then the 'call' statement invokes 'Seq', which is executed as follows:

1. Start the Second Execution of 'Seq'. A second activation region is created for 'Seq' and storage for the automatic variable named 'i' is allocated in it. (Now there are two variables named 'i', and the rule for surface recursion must be applied to select one of them.) The value of 'n' is assigned to 'i' in the most recently created activation region for 'Seq' and is printed as the second line of output. Since 'n' is still greater than one, the value of 'n' is reduced by one and then the 'call' statement invokes 'Seq', which is executed as follows:
 1. Start the Third Execution of 'Seq'. A third activation region is created for 'Seq' and 'i' is allocated in it. The value of 'n' is assigned to 'i' in the most recently created activation region for 'Seq' and is printed as the third line of the output. Since 'n' is not greater than one, the reduction of 'n' and the recursive call of 'Seq' are skipped. Now the recursive calls begin to "unwind".
 2. Complete the Third Execution of 'Seq'. The value of 'i' in the most recently created activation region for 'Seq' is printed as the fourth line of output. The third activation region for 'Seq' is discarded.
2. Complete the Second Execution of 'Seq'. The value of 'i' in the most recently created activation region for 'Seq' is printed as the fifth line of output. (Of course "most recently created" refers only to those activation regions that still exist.) The second activation region for 'Seq' is discarded.
2. Complete the First Execution of 'Seq'. The value of 'i' in the activation region for 'Seq' is printed as the sixth line of output. The activation region for 'Seq' is discarded.
3. Complete Execution of 'R1'. The activation region for 'R1' is discarded.

In order to see how activation regions are handled during surface recursion, consider the following diagram of storage just before the start of the third execution of 'Seq':



At this point in program execution, there are two activation regions for 'Seq'. According to the rule for surface recursion, a reference to 'i' uses the one marked "activation 2".

EXAMPLE OF RECURSION WITH AN ARGUMENT

The preceding example program, named 'R1', can be simplified by transmitting an argument to 'Seq'. The new version is:

```
R2:  proc;
      dcl (sysin,sysprint) file;
      dcl n fixed;
      get list(n);
      call Seq(n);
Seq:  proc(i);
      dcl i fixed;
      put skip list(i);
      if i > 1 then call Seq(i-1);
      put skip list(i);
      end;
      end;
```

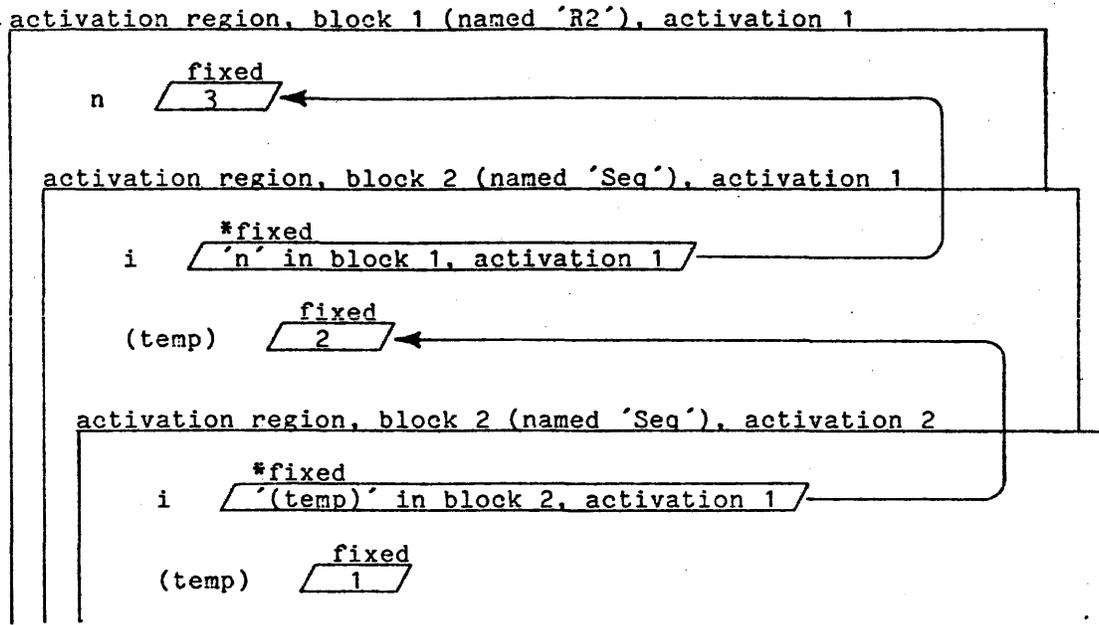
This program is "simplified" not only because it is shorter, but also because, for an experienced programmer, it shows more clearly how the procedure 'Seq' depends on the environment in which it is called.

For each recursive invocation of 'Seq', the program uses two system temporaries to transmit the argument. The temporaries play different roles, as follows:

- A 'fixed' temporary is used to hold the value of the argument 'i-1' in the second call. In Multics, the argument temporary is allocated in the activation region of the calling procedure; therefore, the (n-1)th activation region contains the argument temporary for the nth invocation of 'Seq'.
- A 'pointer' temporary is used to link the parameter to the argument. A reference to the parameter 'i' designates this parameter temporary, and the temporary, in turn, designates the storage unit for the argument. The parameter temporary is allocated in the activation region for the calling procedure; therefore, the (n-1)activation region of 'Seq' contains the parameter temporary for the nth invocation of 'Seq'.

Observe that no argument temporary is required for the first call of 'Seq' because the argument 'n' is transmitted by-reference, and the parameter temporary thus points directly to 'n'.

In order to see how arguments are handled during surface recursion, consider the following diagram just before the start of the third execution of 'Seq':



The storage unit for the parameter, 'i', is given the data type '*fixed' to show that it holds a pointer to a 'fixed' storage unit. Because 'i' is a parameter, a reference to 'i' is interpreted as a reference to the storage unit designated by the pointer. The arrows in the diagram show how each parameter designates and argument storage unit. The storage unit designated '(temp)' is a system temporary used for the value of 'i-1' in the second 'call' statement in the program.

EXAMPLE OF CHAINED RECURSION

For an example of chained recursion, consider one again the program 'R1' that was given as the first example of recursion. A program that produces the same output is:

```
R3:  proc;
      dcl (sysin,sysprint) file;
      dcl n fixed;
      get list(n);
      call Seq;
Seq:  proc;
      dcl i fixed;
      i = n;
      put skip list(i);
      call Test;
      put skip list(i);
      end;
Test: proc;
      if n > 1 then
        do;
          n = n-1;
          call Seq;
        end;
      end;
end;
```

There is no direct recursion in this version of the program: 'Seq' does not contain a call on itself and 'Test' does not contain a call on itself. However, 'Seq' calls 'Test', and 'Test' calls 'Seq', so recursion does occur.

RECURSIVE PROGRAM

The following program contains a recursive procedure, 'COMP', that is a rudimentary translator of expressions:

```
P:  proc;
    dcl sysprint file;
    call COMP("(((a-b)/c)*(d+(e+f)))",1,0);
    put skip;
COMP: proc(S,i,n) recursive;
    dcl S char(*);
    dcl i fixed;
    dcl n pic"999";
    dcl a char(9) var initial("");
    dcl c char(1);
    dcl k fixed;
    do k = i+1 by 1
        c = substr(S,k,1);
        if c = "("
            then do;
                call COMP(S,k,n);
                a = a||"T"||n;
            end;
        else if c = ")"
            then do;
                n = n+1;
                put skip list("T"||n||"="||a||");
                i = k;
                return;
            end;
        else a = a||c;
    end;
end;
```

The program is set up for a test run of the 'COMP' procedure. The main procedure invokes 'COMP' on an expression that is given as a character string; namely:

```
((a-b)/c)*(d+(e+f))
```

The output of the recursive execution of 'COMP' is:

```
T001=a-b;
T002=T001/c;
T003=e+f;
T004=d+T003;
T005=T002*T004;
```

Thus each operator in the given expression is compiled into a PL/I statement.

The following assertions about variables help to explain the execution of the procedure:

- S contains the parenthesized expression that is to be translated into a sequence of assignment statements. Its value does not change throughout the program. In fact, for each invocation of 'COMP', 'S' designates the same storage unit; namely the temporary that is allocated for the argument of the first invocation.
- i designates the '(' that begins the parenthesized expression within 'S' that is to be translated by a particular invocation.
- n contains the number of the most recently used name in the series 'T001', 'T002', and so on.
- a contains the character string that represents the assignment statement that is being formed by the current invocation of the procedure. The variable is 'automatic'; otherwise the procedure would not work. For example, the initial invocation of the procedure proceeds as follows:

```
set 'a' to ""
invoke 'COMP' to compile "((a-b)/c)"
set 'a' to "T002"
invoke 'COMP' to compile "(d+(e+f))"
set 'a' to "T002*T004"
```

Clearly the intermediate values of 'a' would be lost if 'a' were not allocated for each new invocation.

- c contains the current character in the scan of the given expression. The variable is 'automatic', but need not have been because there is no recursive call between its setting and its last use.
- k designates the current character for a given execution of the body of the 'do' group; it controls the scan of the parenthesized expression. It is set to 'i+1' in order to skip the initial '('.

In order to study the procedure, simulate an invocation until the recursive call is reached. Instead of following that call, assume that it returns the correct result and complete the simulation of the current invocation.

General Recursion

A recursive program uses general recursion if any of the following statements are true:

- The program obtains the 'pointer' value that designates an 'automatic' variable during one activation of a block and then use that 'pointer' value to locate the 'automatic' variable during a later activation of the same block.
- The program obtains a statement address value ('entry', 'format', or 'label' value) during one activation of a block and then uses the value to locate a statement during a later activation of the same block.
- The program establishes an 'on' unit during one activation of a block and then signals the 'on' unit during a later activation of the same block.

In these statements, the phrase "activation of a block" refers either to the invocation of a procedure or the execution of a 'begin' block.

At certain points in the execution of a recursive program, a name must be interpreted as a reference to a storage unit in an activation region. However, the interpretation of a name given in Section VI, "Declarations" yields only the declaration of the name and the block in which its declaration is established. An additional rule is required to select one of the several activation regions for the given block. The general rule is complicated, and depends on extensions to the interpretation of several features of PL/I.

Rules for the interpretation of general recursion are given here. The rules introduce extra steps into the execution of a program. These steps are essential if the program uses general recursion; otherwise, they have no effect on the final outcome. Thus the rules can be applied to all programs or just to those that use general recursion. In fact, the Multics implementation of PL/I does not determine the recursive properties of a program but rather applies these rules to all programs. In contrast, a programmer who is studying a given program should probably ignore these rules unless the program uses general recursion.

General recursion requires special treatment of certain address values. The discussion that follows begins with the description of the activation index, which is used in an address value to designate an activation region. The discussion continues with a few paragraphs that explain the use of a 'pointer' value that has an activation index. The remainder of the discussion is devoted to statement address values: the parent index is introduced, rules for the use and setting of parent indexes are given, and several examples are included.

ACTIVATION INDEXES

A particular activation region may be conceptually designated by adding an activation index to the designation for the block. Consider, for example, the value described by "'X' in block 4, activation 2". Depending on the declaration of 'X', this address value is interpreted as follows:

- If 'X' is an 'automatic' variable name, then the value is a 'pointer' value that designates the storage unit named 'X' in the second activation of block 4.
- If 'X' is a statement name, it designates the statement named 'X' immediately contained in block 4 and, further, it specifies the interpretation of that statement within the second activation of block 4.

The way in which these values are generated and used is described later in this discussion of general recursion.

In an implementation of PL/I it is certainly not necessary to represent an address value as "'LAB' in block 4, activation 2"; such a representation is convenient for discussion but not for implementation. All that is necessary is a representation that conveys the essential information. In the Multics implementation of PL/I, Multics pointers are used. A Multics pointer plays the role of a hardware address, and it can designate any location in Multics storage. The following representations are used:

- A 'pointer' value is represented by a Multics pointer that designates a storage unit.
- A statement address value is represented by a pair of Multics pointers. The first pointer designates the beginning of the code for a statement. The second pointer, the activation pointer, designates the beginning of a stack frame.

Thus, for reasons of efficiency, these values are given quite different representations in Multics.

POINTERS

A 'pointer' value that points to storage in an activation region can be formed by the application of the 'addr' function to a reference to an 'automatic' variable. The argument of the 'addr' function is interpreted to locate a specific variable in a specific activation region; then that information is expressed as a 'pointer' value. The storage may later be accessed through the 'pointer' value even though the activation region may no longer be accessible in other ways. The following example illustrates this case.

Example of Pointers

As an example of the formation and use of a 'pointer' value when general recursion is present, consider the following program:

```
Q1:  proc;
      dcl sysprint file;
      dcl n fixed static initial(1);
      dcl i fixed;
      dcl ib based(ip);
      dcl ip pointer static;
      i = n;
      if n = 1 then ip = addr(i);
      put skip list(i,ib);
      n = n+1;
      if n < 3 then call Q1;
      end;
```

The output of this program is:

```
1      1
2      1
3      1
```

Each number on each line of this output represents the value of a variable named 'i'. However, the first number in a line is taken from the variable that is allocated in the most recently created activation region, whereas the second number is taken from the variable that is allocated in the first (and least recently created) activation region.

The program is invoked three times, the first time by a Multics command and the second and third time by itself, recursively. During the first, second, and third invocations, the value of 'n' is '1', '2', and '3', respectively; and during each invocation, the current value of 'n' is assigned to a variable named 'i' that is allocated in the first, second, or third activation region. The important feature of the program is the statement:

```
if n = 1 then ip = addr(i);
```

The assignment to 'ip' is performed only during the first execution of the program and therefore the value of 'ip' designates the same allocation of 'i' throughout the program, namely "i' in block 1 (named Q2), activation 1".

Suppose the 'if' statement is changed to the following:

```
if n < 3 then ip = addr(i);
```

Then the output of the program is:

1	1
2	2
3	2

Suppose, on the other hand, that the 'if' statement is changed to the following:

```
if n = 2 then ip = addr(i);
```

Then the program is invalid because a reference is made to 'ip' (during the first invocation) before a value is assigned to 'ip' (during the second invocation).

PARENT DESIGNATORS

Each activation region must contain a parent designator called the static link. The parent designator in an activation region for a given block designates the most recent activation regions for the block that immediately contains the given block. The parent designator in an activation region for an external procedure has no purpose and can be ignored. A parent designator is used in interpreting a program but is not directly accessible from the program.

The parent designators are used in the definition of a current activation region for each block that contains the most recently activated block. The definition is as follows:

- The current activation region for the most recently activated block is the most recently created activation region.
- The current activation region for a block that immediately contains a given block is the activation region designated by the parent designator of the given block.

As an example of the use of this definition, let the most recently activated block of a program be called B1; let the block that immediately contains B1 be called B2; let the block that immediately contains B2 be called B3; and let B3 be an external procedure. Then the current activation region for each of these blocks is determined as follows:

- The most recently created activation region for B1 is the current activation region for that block.
- Suppose that the parent designator in the current activation region for B1 designates the third activation region for B2; then the latter is the current activation region for B2.
- Suppose that the parent designator in the current activation region for B2 designates the first activation region for B3; then the latter is the current activation region for B3.

Thus the parent designators form a sequence of activation regions that begins with the most recently activated block and ends at the containing external procedure.

Observe that the blocks for which current activation regions are defined are exactly those blocks whose declared names can be referenced in the most recently activated block. The purpose of defining the current activation region is to resolve ambiguities that arise in the interpretation of certain references to those names. Such ambiguities arise in two cases: the interpretation of activation variable references and the evaluation of statement address constant names. Discussion of these cases follows.

Activation Variable References

Certain variables designate storage units that are allocated in an activation region. There are three such cases:

- An 'automatic' variable reference designates a variable that is allocated in an activation region.
- A 'defined' variable reference may or may not designate a variable that is allocated in an activation region; however, each extent (array bound, maximum string length, or area size) for the variable is contained in a system temporary that is allocated in an activation region.
- A 'parameter' variable reference designates a parameter pointer that is contained in a system temporary that is allocated in an activation region.

Except in the case of a 'parameter' variable reference, the designated storage is allocated in an activation region that is associated with the block in which the variable name is declared. If there is more than one such activation region, then the choice is resolved by selecting the current activation region for the block.

Statement Address Constant References

The evaluation of a given statement address constant reference yields an 'entry' value, a 'format' value, or a 'label' value. Earlier in this discussion, it was noted that such values must include an activation index. That activation index is chosen so that it designates the current activation region for the block in which the name in the given reference is declared.

SETTING THE PARENT DESIGNATOR

An activation region is created when a 'procedure' block, a 'begin' block, an 'on' unit, or a 'format' statement is executed. For general recursion, the activation region must include a parent designator. Each of the four cases is described in the following paragraphs.

'procedure' Block

Part of the invocation of a procedure is the evaluation of an 'entry' reference. The resulting 'entry' value includes an activation region designator, and it is this designator that is used as the parent designator in the activation region that is created for the execution of the procedure.

The Multics implementation of PL/I optimizes by avoiding the creation of a block activation region for certain procedures; however, the interpretation of the program is not changed. A programmer must be aware of this practice only when he uses Multics to examine stack frames directly. The compiler prints a message for each procedure that is compiled without an activation region.

As an example of procedure invocation in the presence of general recursion, consider the following program:

```
Q2:  proc;
      dcl sysprint file;
      dcl n fixed static init(0);
      dcl i fixed;
      dcl ev entry variable static internal;
      n = n+1;
      i = n;
      if n = 1 then ev = S;
      put skip list(i);
      call ev;
      if n < 3 then call Q2;
S:    proc;
      put list(i);
      end;
      end;
```

The output of this program is:

```
1      1
2      1
3      1
```

Each line of the output is produced by outputting the value of 'i' twice; once by a statement in the main procedure and once by a statement in 'S'. The first value of 'i' is always from the most recent activation region for 'Q2', whereas the second value of 'i' is from the first (least recent) activation of 'Q2'.

The following aspects of the program just given are particularly interesting:

- A value is assigned to the 'entry' variable only once; namely, during the first activation of 'Q2'. Therefore, the value of 'ev' is equivalent to "S' in block 1 (named Q2), activation 1".
- The procedure 'S' is invoked three times, once in each activation of 'Q2'. Each time, the 'entry' value has the activation index "activation 1", as just noted. Therefore, the parent designator in the activation region for 'S' is always equivalent to "block 1, activation 1".
- Each time 'i' is evaluated within 'S', the reference is to the current activation region for 'Q2'. That activation region is specified by the parent pointer in the activation region for 'S', as just described, and is always activation 1.

'begin' Block

When a 'begin' block that is not an 'on' unit is executed, the parent designator in the activation region that is created for the execution of the block is set to designate the most recent activation region of the block that immediately contains the given 'begin' block.

'format' Statement

The invocation of a 'format' statement is similar to the invocation of a 'procedure' block, but a 'format' statement is used in a very restricted way, as described in Section XIV, "Stream Input/Output." Part of the invocation of a 'format' statement is the evaluation of a 'format' reference. The resulting 'format' value includes an activation region designator, and that designator is used as the parent designator in the activation region that is created for the execution of the procedure.

In order to reduce execution cost, the Multics implementation of PL/I avoids the creation of a block activation region for a 'format' statement. This optimization technique does not change the interpretation of a program, and it can be ignored by a programmer who remains within the framework of the PL/I language.

As an example of the invocation of a 'format' statement, consider the following program:

```
Q3: proc;
    dcl sysprint file;
    dcl n fixed static init(0);
    dcl i fixed;
    dcl fv format variable static;
    n = n+1;
    i = n;
    if n = 1 then fv = F;
    put skip edit(2,2)(r(F),r(fv));
    if n < 3 then call Q3;
F:   format(e(12,i));
    end;
```

The output of this program is:

```
2.0e+000  2.0e+000
2.00e+000 2.0e+000
2.000e+000 2.0e+000
```

Each line of the output is produced by outputting the value '2' twice. Each output value is edited by the format statement 'F', but the format statement is invoked in two different ways and therefore the parameter 'i' (which determines the number of digits after the decimal point) is evaluated in different activations. Specifically,

- The reference 'r(F)' always causes the format statement to be evaluated within the most recent activation of 'Q3' so the value of 'i' is '1', '2', and '3'.
- The reference 'r(fv)' causes the format statement to be evaluated within the activation specified in the 'format' variable. Observe that the format variable is set only in the first activation of 'Q3'. Therefore all the invocations of the 'format' statement use the first activation region of 'Q3' and 'i' is '1'.

'on' Unit

When an 'on' statement is executed, it establishes (but does not execute) an 'on' unit for a given condition. As part of this action, the designation of the 'on' unit is associated with the given condition and saved. For the purposes of general recursion, the designation of the most recently created activation region for the block that immediately contains the 'on' statement is also associated with the condition and saved.

When a condition is signalled for which the program has established an 'on' unit, the most recently established 'on' unit is executed. As part of that execution, an activation region for the 'on' unit is created (regardless of whether the 'on' unit is a 'begin' block or a single statement). The parent pointer in that activation region is the activation region designator that was saved when the 'on' unit was established.

As an example of the invocation of an 'on' unit, consider the following program:

```
Q4: proc;
    dcl (sysin,sysprint) file;
    dcl n fixed static init(0);
    dcl i fixed;
    dcl k fixed;
    dcl conv cond;
    n = n+1;
    i = n;
    if i = 1 then on conv put skip list("***",i);
    get list(k);
    put list(i);
    if i < 3 then call Q4;
end;
```

Suppose the input for this program is:

```
6    5    four
```

Then the output of the program is:

```
1
2
***      1
3
```

Since the 'on' condition is established during the first invocation of 'Q4', its associated activation index is "activation 1". During the third activation of 'Q4', the unacceptable input item, 'four', is encountered, and the 'conversion' condition is signalled. The 'on' condition is invoked, but the parent designator in its activation region is equivalent to "block 1 (Q4), activation 1". For that reason, the 'on' unit prints out a '1'.

'goto' STATEMENT

Part of the execution of a 'goto' statement is the evaluation of a 'label' reference. The resulting 'label' value includes an activation region designator. The 'goto' statement causes exits from any block activations that are more recent than the block activation that is associated with the activation region designated by the 'label' value; then execution resumes at the designated statement.

As an example of the interpretation of a 'goto' statement in the presence of general recursion, consider the following program:

```
Q5: proc;
    dcl sysprint file;
    dcl n fixed static init(0);
    dcl i fixed;
    dcl lv label variable static;
    n = n+1;
    i = n;
    if i = 1 then lv = EXIT;
    if i < 3 then call Q5;
    put skip list(i);
    goto lv;
EXIT: end;
```

The output of this program is:

3

The 'label' variable 'lv' is set during the first invocation of the procedure to a value equivalent to "'EXIT' in block 1 (named Q5), activation 1". During activation 3 of the procedure, the 'goto' statement is executed for the first time. The effect is an exit from activations 3 and 2 of 'Q5' and a transfer to the statement labeled 'EXIT'.

As a rather different example of the interpretation of a 'goto' statement, consider the following program:

```
Q6: proc;
    dcl sysprint file;
    dcl n fixed static init(0);
    dcl i fixed;
    dcl ev entry variable static int;
    n = n+1;
    i = n;
    if n = 1 then ev = S;
    if n < 3 then call Q6;
    put skip list(i);
    call ev;
S:    proc;
        goto EXIT;
        end;
EXIT: end;
```

Once again the output of this program is a single line:

3

The statement address constant reference, 'EXIT', in the 'goto' statement is evaluated during the third activation of the procedure 'Q6' in which 'EXIT' is declared. However, its value is not "'EXIT' in block 1 (named Q6), activation 3". Instead, when 'S' is invoked (for the first and last time), the parent designator in its activation region is "block 1 (named Q6), activation 1". Therefore, the current activation region of 'Q6' during the execution of 'S' is the first activation region and the value of 'EXIT' is "'EXIT' in block 1, activation 1". Just as in the previous example, the 'goto' statement causes an exit from activation 3 and 2 of 'Q5' and a transfer to the statement labeled 'EXIT' in activation 1.

SECTION XIII

CONDITION HANDLING

Sometimes a program instructs the processor to perform an action that cannot be performed. In this situation, it is said that an exceptional condition, or simply a condition, has occurred. In some cases, the occurrence of a condition is an error, as with an attempt to divide by zero. In other cases, the occurrence of a condition communicates the state of an external file, as with an attempt to read beyond the last record of a file. In all cases, some action must be specified as an alternative to the action that cannot be performed.

The PL/I processor can handle exceptional conditions by itself; in that case, it will, for most conditions, print an error message and abort the program. However, the PL/I language permits the programmer to supply a programmed response, called an 'on' unit, to the condition. In fact, the facilities for handling conditions are quite elaborate. These facilities are the subject of this section.

PRINCIPAL FEATURES OF CONDITION HANDLING

To handle a condition, the programmer first declares the condition by means of a 'declare' statement. Next, he enables the detection of the condition by means of a condition prefix unless the condition is already enabled. Finally, he establishes an 'on' unit for the condition by means of an 'on' statement. If the condition is signalled after the execution of the 'on' statement, the established 'on' unit established by the 'on' statement is executed.

As an example of condition handling, consider the following program:

```
P:  proc;
    dcl endfile cond;
    dcl (sysin,sysprint) file;
    dcl x float;
    on endfile(sysin) goto exit;
loop: get list(x);
      put list(x,x**2);
      goto loop;
exit:  end;
```

This program illustrates the handling of the 'endfile' condition for the system input stream. First, the program declares 'endfile' as a condition name. The 'endfile' condition is always enabled, so the program does not enable it. Then the program establishes an 'on' unit for the condition reference 'endfile(sysin)'. When the 'get' statement encounters an end-of-file, the 'endfile' condition is signalled for 'sysin' and the established 'on' unit for that condition reference is executed. In this program, the established 'on' unit transfers control to the end of the program.

CONDITIONS

The condition handling facility of PL/I can be applied to two kinds of conditions, namely:

Language-defined conditions
Programmer-defined conditions

The language-defined conditions are defined as part of the PL/I language and detected by the PL/I processor during its execution of a program. The programmer-defined conditions are defined by the programmer and signalled by the program.

Language-Defined Conditions

The language-defined conditions are divided into four categories, as follows:

Computational
Storage
Termination
Input/Output

The following paragraphs define each category and then give a brief description of each condition in the category. The description ends with a reference to the section of this manual in which a complete description can be found.

COMPUTATIONAL CONDITIONS

The computational conditions occur when errors are detected during the conversion of values or the evaluation of expressions. The computational conditions are the only conditions that can be enabled and disabled. A description of the enabling and disabling of conditions is given later in this section.

The condition reference for a computational condition is the condition name or its abbreviation. The computational conditions are summarized in the following list.

<u>Condition Reference</u>	<u>Description</u>
conversion conv	occurs when an invalid character string or pictured value is converted to an arithmetic or bit-string value, as described in Section IV "Value Conversion."
fixedoverflow fofl	occurs when the result of a binary fixed-point computation exceeds 71 digits, as described in Section IV, "Value Conversion."
overflow ofl	occurs when the result of a floating-point computation is too large to be represented, as described in Section IV, "Value Conversion."
size	occurs when a value is assigned to a fixed-point target and the precision cannot accommodate the magnitude of the number, as described in Section IV, "Value Conversion."

stringrange strg	occurs when a designated substring is not completely contained in the string argument of a 'substr' function reference or pseudo-variable, as described in Section IX, "Operations."
stringsize strz	occurs when a character-string or bit-string is converted for a target whose length cannot accommodate the value, as described in Section IV, "Value Conversion."
subscriptrange subrg	occurs when the value of a subscript exceeds the bounds of the dimension to which it corresponds as described in Section VIII, "Expressions."
underflow ufl	occurs when the result of a floating-point computation is too small to be represented, as described in Section IV, "Value Conversion."
zerodivide zdiv	occurs when the divisor of a division operator is zero, as described in Section VIII, "Expressions."

STORAGE CONDITIONS

The storage conditions occur when the capacity of PL/I storage is exceeded. In some cases, an 'on' unit can free storage and thus recover from a storage condition. If a storage condition is signalled and the established 'on' unit does not free sufficient storage, the condition is signalled again.

The condition reference for a storage condition is the condition name. The storage conditions are summarized in the following list.

<u>Condition Reference</u>	<u>Description</u>
area	occurs when an attempt is made to allocate storage in an area variable that cannot supply the storage, as described in Section VII, "Storage Management."
storage	occurs when the Multics stack segment is about to overflow or when the "system storage" in which 'controlled' and 'based' variables are allocated is full, as described in Section VII, "Storage Management."

TERMINATION CONDITIONS

The termination conditions occur when execution of a program is complete.

The condition reference for a termination condition is the condition name. The termination conditions are summarized in the following list.

<u>Condition Reference</u>	<u>Description</u>
finish	occurs when the process is terminated, as described later in this section.
error	occurs as a result of a fatal error in program execution, as described later in this section.

INPUT/OUTPUT CONDITIONS

The input/output conditions occur when an input/output operation requires special attention. Some of the input/output conditions indicate errors, but other conditions communicate a valid change in the status of the data set.

The condition reference for an input/output condition consists of the condition name followed by a parenthesized file reference. The input/output conditions are summarized in the following list.

<u>Condition Reference</u>	<u>Description</u>
endfile(fr)	occurs when an input statement attempts to read past the end-of-file on the referenced file, as described in the Sections XIV and XV, "Stream Input/Output" and "Record Input/Output," respectively.
endpage(fr)	occurs when a 'put' statement attempts to output a line that does not fit on the current page, as described in Section XIV, "Stream Input/Output."
key(fr)	occurs when a 'key' option specifies a key that does not exist in the referenced file or when a 'keyfrom' option specifies a key that already exists in referenced file, as described in Section XV, "Record Input/Output."
name(fr)	occurs when an assignment is read from the referenced stream and the variable name does not match any of the variable names in the data list of the controlling 'get' statement, as described in Section XIV, "Stream Input/Output."
record(fr)	occurs when a 'read' statement reads a record from the referenced file that is not equal in size to the variable given in the 'into' option, as described in Section XV, "Record Input/Output."
transmit(fr)	occurs when the data cannot be reliably transmitted between the referenced file and storage, as described in Section XIV and XV, "Stream Input/Output" and "Record Input/Output," respectively.
undefinedfile(fr)	occurs when an attempt to open the referenced file is unsuccessful, as described in Sections XIV and XV, "Stream Input/Output" and "Record Input/Output," respectively.

Programmer-Defined Conditions

The condition handling facility of PL/I can be used to define new conditions that are designed for the particular needs of a given programming application. By defining new conditions, a software system written in PL/I, such as Multics, can provide its sub-systems with the capability of handling conditions.

The Multics operating system, which is written in PL/I, makes use of programmer-defined conditions. For example, the Multics system defines the condition `program_interrupt` to allow commands to respond to interruptions from the terminal. This condition is signalled in the following way:

- When the user presses the interrupt button on his terminal, the quit condition is signalled.
- The Multics handler for the quit condition prints 'QUIT', aborts any pending terminal input/output activity, restores the mode of user-io and the standard I/O attachments to the default settings, saves the current stack history, and establishes a new command level.
- If the user then types the command 'pi' on his terminal, the `program_interrupt` condition is signalled.

If the interrupted command has established an 'on' unit for the `program_interrupt` condition, that 'on' unit is executed; otherwise, the 'on' unit provided by the Multics system is executed.

The text editor `edm` is a Multics command that makes use of the `program_interrupt` condition. The 'on' unit provided by `edm` for this condition concludes the interrupted activity and calls for a request from the user's terminal. Thus, a user can terminate an `edm` printout by pressing the interrupt button on his terminal and then typing 'pi'. The `edm` system, in response to this action, terminates the printing, types 'Edit', and waits for the user request.

CONDITION REFERENCES

In the discussion of conditions, a condition reference was given for each condition. The condition reference is used in the 'on' statement, the 'revert' statement, and the 'signal' statement to designate a condition. There are two kinds of condition references corresponding to the two kinds of conditions.

Language-Defined Condition References

A language-defined condition reference must have one of the following forms:

lcn

lcn(fr)

where lcn is a language-defined condition name and fr is the file reference. The language-defined condition name must be declared with the 'condition' attribute. For the first form the language-defined condition name must be one of the following identifiers:

area	fixedoverflow	storage	subscriptrange
conversion	fofl	stringrange	subrg
conv	overflow	strg	underflow
error	ofl	stringsize	ufl
finish	size	strz	zerodivide
			zdiv

For the second form, the language-defined condition name must be one of the following identifiers:

endfile	key	record	undefinedfile
endpage	name	transmit	undf

The file reference must be a reference that yields a scalar file value.

Programmer-Defined Condition References

A programmer-defined condition reference must have one of the following forms:

condition(pcn) cond(pcn)

condition pcn cond pcn

where pcn is the programmer-defined condition name. The programmer-defined condition name must be declared with the 'condition' attribute and must not be one of the identifiers listed as language-defined condition names in the preceding definition of language-defined condition references. The four forms of programmer-defined condition reference are equivalent to one another.

DECLARATION OF CONDITION NAMES

Each condition name used in a program should be declared, as follows:

```
del cn cond;
```

where cn is the condition name. The condition name can be either a language-defined condition name or a programmer-defined condition name.

The only attribute that can be used with the 'condition' attribute is the scope attribute. In Multics, every condition name must have 'external' scope. Since this is the default scope for condition names, the scope attribute is omitted from the 'declare' statement.

ENABLING AND DISABLING CONDITIONS

The language-defined computational conditions can be enabled and disabled. When a condition that is enabled occurs, it is signalled and handled. When a condition that is disabled occurs, it may or may not be signalled.

The occurrence of a language-defined condition is detected either by the computer hardware or by additional code compiled into the program to test for the occurrence of the condition. When a condition that requires additional code for its detection is disabled, the resulting program requires less storage and executes more efficiently than a program for which the condition is enabled. However, if a disabled condition occurs in such a program, the program is invalid and the results of its continued execution are undefined.

Condition Prefixes

Condition prefixes are used to enable or disable language-defined computational conditions. The condition prefix must precede the label prefixes (if any) of a statement. The form of the condition prefix list is as follows:

```
(cpn, ...)
```

where cpn, ... is a sequence of condition prefix names separated by commas. The possible condition prefix names are enabling prefix names and disabling prefix names. An enabling prefix name is any of the computational condition names or their abbreviations. A disabling prefix name is the characters 'no' followed by an enabling prefix name. The complete list of condition prefix names is:

<u>Enabling Prefix Names</u>		<u>Disabling Prefix Names</u>	
<u>Name</u>	<u>Abbr.</u>	<u>Name</u>	<u>Abbr.</u>
conversion	conv	noconversion	noconv
fixedoverflow	fofl	nofixedoverflow	nofofl
overflow	ofl	nooverflow	noofl
size		nosize	
stringrange	strg	nostringrange	nostrg
stringsize	strz	nostringsize	nostrz
subscriptrange	subrg	nosubscriptrange	nosubrg
underflow	ufl	nounderflow	noufl
zerodivide	zdiv	nozerodivide	nozdiv

A condition prefix can begin any PL/I statement except the 'declare', 'default', or 'entry' statements.

Scope of a Condition Prefix

A condition prefix applies to the statement to which it is attached. For most statements, the scope of the condition prefix is the entire statement; however, there are important exceptions, as follows:

<u>Statement</u>	<u>Scope</u>
'procedure', 'begin'	The condition prefix applies to all statements in the block that begins with the 'procedure' or 'begin' statement except those statements that lie within the scope of another condition prefix for the same condition.
'if'	The condition prefix applies only to the test, which lies between the keywords 'if' and 'then'; it does not apply to the consequences.
'do'	The condition prefix applies only to the 'do' statement; it does not apply to the remaining statements of the 'do' group.
'on'	The condition prefix applies only to the condition reference; it does not apply to the 'on' unit that follows the condition reference.
'format'	The condition prefix applies only to calculations performed in the evaluation of the format specification list; it does not apply to associated 'get' or 'put' statements or to any other 'format' statements.

As an example of the determination of the scope of a condition prefix, consider the following statement:

```

if a(i,j) = 1
  then a(i,j) = b(k,m);
  else a(i,j) = c(p,n);

```

To enable the 'subscriptrange' condition for the entire 'if' statement, it is necessary to apply the condition prefix to both the test and the consequences, as follows:

```
(subrg):
    if a(i,j) = 1
        then
    (subrg):      a(i,j) = b(k,m);
        else
    (subrg):      a(i,j) = c(p,n);
```

A different way of enabling the condition for the entire 'if' statement is to enclose the statement in a 'begin' block and then to apply the condition prefix to the block, as follows:

```
(subrg):
    begin;
        if a(i,j) = 1
            then a(i,j) = b(k,m);
            else a(i,j) = c(p,n);
        end;
```

In practice, a condition prefix is usually applied to a procedure block and thus the condition is enabled or disabled for the entire procedure.

Default Enabling and Disabling

If a statement does not lie within the scope of a condition prefix for a given condition, then the statement is interpreted under the default enabling or disabling for the given condition, as follows:

<u>Enabled by Default</u>	<u>Disabled by Default</u>
conversion	size
fixedoverflow	stringrange
overflow	stringsize
underflow	subscriptrange
zerodivide	

These defaults are dictated by considerations of efficiency. The conditions that are detected by the hardware are enabled by default and the conditions that require additional instructions in the compiled program for their detection are disabled by default.

Example of Enabling and Disabling

As an example of enabling and disabling, consider the following program:

```
(subrg):
P:  proc;
    dcl (a,b,c)(50,100) fixed;
    dcl (i,j) fixed;
    ...
    b(i,j) = a(i,j)**2;
(nosubrg):
    c(i,j) = b(i,j)**2;
(nosubrg,size):
    c(j,i) = c(i,j)**2;
    ...
end;
```

The 'subscriptrange' condition, which is disabled by default, is enabled for most of this program because of the '(subrg):' condition prefix on the 'procedure' statement. However, 'subscriptrange' is disabled for the two assignments to the array 'c'. Furthermore, the 'size' condition, which is disabled by default, is enabled for the last assignment statement.

ESTABLISHING AND REVERTING 'on' UNITS

A programmer has a choice between permitting the system to handle a signalled condition or writing a statement to handle the condition in a special way. The construct supplied by the programmer to handle the condition is an 'on' unit. An 'on' unit is established for a condition by the execution of an 'on' statement. When the condition is signalled, the established 'on' unit is executed.

In some cases, a single 'on' unit can be established to apply to all signals of a given condition. In other cases, the handling of the condition depends on the statement in which the condition occurs. In the second case, several 'on' units are established for a condition by the execution of several 'on' statements.

The establishment of an 'on' unit is associated with the current block activation. For a given block activation, at most one 'on' unit can be established at any time for each evaluated condition reference. Thus, for example, "if an 'on' statement is executed for 'endfile(sysin)' and an 'on' unit for 'endfile(sysin)' is already established for the current block, the previously established 'on' unit is removed from the set of established 'on' units for the block before the current 'on' unit is added to the set. The execution of a 'revert' statement removes an established 'on' unit from the set.

The rules for determining the established 'on' unit for a signalled condition are given later in this section in "Occurrence and Signalling".

'on' Statement

An 'on' statement has one of the following forms:

on cr-list ou;

on cr-list snap ou;

where cr-list is a list of condition references separated by commas and where ou the 'on' unit. The 'on' unit must be one of the following:

- A 'begin' block; that is, a 'begin' statement, followed by a sequence of statements, followed by an 'end' statement.
- A restricted independent statement; that is, one of the following statements:

storage management: 'allocate' and 'free'

flow of control: 'goto' and the null statement

procedure invocation: 'call'

condition handling: 'signal'

input/output: 'open' and 'close'

stream input/output: 'put' and 'get'

record input/output: 'read', 'write', 'delete', 'rewrite' and 'locate'

- the keyword 'system'

An 'on' unit can contain a 'return' statement only if the 'return' statement is contained in a 'procedure' block that is contained in the 'on' unit.

When an 'on' statement is executed, all of the conditions in cr-list are evaluated and the 'on' unit for each evaluated condition reference is established. Each evaluated condition reference and a designator for the 'on' unit are added to the set of established 'on' units for the current block activation. The 'on' unit in the 'on' statement is not executed until the condition is signalled. If the 'on' unit consists of the keyword 'system', the default 'on' unit is established.

The 'snap' option specifies that debugging information is to be provided when the condition is signalled. In Multics PL/I, the action taken depends on the type of process. In an interactive process, 'snap' causes a call to be made to the probe command, just prior to the execution of the 'on' unit. In an absentee process, 'snap' causes a call to be made to the trace_stack command.

'revert' Statement

The 'revert' statement has the following form:

```
revert cr, ...;
```

where cr is a condition reference.

When a 'revert' statement is executed, the condition reference is evaluated and the 'on' unit for that evaluated condition reference is removed from the set of established 'on' units for the current block activation.

OCCURRENCE AND SIGNALLING OF CONDITIONS

A condition occurs as a result of the interpretation of the program by the PL/I processor. If a condition occurs when it is enabled, the condition is signalled and the established 'on' unit is executed. If a condition occurs when it is disabled, the program is invalid and the results of its continued execution are undefined.

If a 'signal' statement for a condition is executed when the condition is enabled, the condition is signalled and the established 'on' unit executed. If the 'signal' statement is executed when the condition is disabled, no action is taken and control passes to the next statement.

Occurrence of Conditions

As an example of the occurrence of a condition, consider the following program:

```
P:  proc;
    dcl (a,b,c) fixed;
    a = 1;
    b = 5 - .5 * a;
    c = a / b;
    end;
```

The 'zerodivide' condition occurs when the third assignment statement is interpreted by the PL/I processor. The 'zerodivide' condition is enabled by default, so the condition is signalled. No 'on' unit is established for the condition, so the default 'on' unit is executed and the program halts with an error message.

The detailed description in this manual for each condition states when the condition can occur. Some conditions, however, are signalled from support subroutines when an error is detected or a limitation is exceeded. In these cases, the condition is signalled, even if it is disabled in the PL/I program, because the signal originates from a support subroutine in which the condition is enabled. The following conditions belong to this category;

```
area
error
fixedoverflow
overflow
size
storage
stringsize
zerodivide
```

These conditions are described as conditions that can occur at anytime during the execution of the program.

As an example of a condition that occurs at an unexpected time, consider the following program:

```
P:  proc;
    dcl n fixed(35) init(200000000);
    dcl x fixed;
    dcl sysprint file;
    x = n;
    put list(x);
    end;
```

The assignment of 'n' to 'x' is invalid since the value of 'n' cannot be represented in the precision of 'x'. The 'size' condition occurs on this assignment. Because the 'size' condition, is disabled by default in the procedure 'P', the condition is not signalled. However, the 'size' condition is enabled in the run-time conversion routines and is signalled when the 'put' statement is executed.

'signal' Statement

The 'signal' statement has the following form:

```
signal cr
```

where cr is a condition reference.

Execution of the 'signal' statement signals the condition if it is enabled. If the condition is disabled, no action is taken.

Determining the Established 'on' Unit

As described earlier in this section under "Establishing and Reverting 'on' Units", each block activation has associated with it a set of established 'on' units. The set of established 'on' units for a block contains at most one established 'on' unit for an evaluated condition reference. When a condition is signalled, the established 'on' unit for that condition is determined in the following way:

1. Let the most recently activated block be the current block.
2. Examine the set of established 'on' units for the current block. If an established 'on' unit for the signalled condition is encountered, take that as the established 'on' unit.
3. If the current block is not the outer block, let the next most recently established block be the current block, and return to step 2. Otherwise, the program does not provide an 'on' unit, so take the default 'on' unit as the established 'on' unit.

Thus, the set of established 'on' units is searched, starting with the most recently activated block and proceeding back through the previously activated blocks.

'on' UNIT

An 'on' unit specifies the action taken when a condition is signalled. The execution of an 'on' unit is similar to the execution of a procedure block. The 'on' unit is activated when the condition is signalled and, if control passes to the end of the 'on' unit, control is returned to the point at which the condition was signalled.

Some PL/I conditions are fatal conditions, namely:

- area (if caused by assignment)
- error
- fixedoverflow
- overflow
- size
- storage (if no additional storage is available)
- stringrange
- subscriptrange
- zerodivide

If an 'on' unit for any of these fatal conditions returns to the point at which the condition was signalled, the program is invalid and the results of its continued execution are undefined.

If an 'on' unit executed as a result of the signalling of a condition during the evaluation of an expression returns to the point at which the signal was detected, the 'on' unit must not allocate, free, or assign a value to any generation of storage accessible at the point where the condition was detected.

If an 'on' unit is executed as a result of the signalling of a condition during the execution of a statement, the 'on' unit must not access the value of the variable changed by the execution of the statement.

Default 'on' Units

If no 'on' unit is established by the program for a condition at the time the condition is signalled, the system default for the 'on' unit is invoked.

In Multics, the following default 'on' units are defined for the listed conditions:

<u>Condition</u>	<u>Default 'on' Unit</u>
name underflow	Writes an error message on the 'error_output' stream and returns control to the point at which the condition was detected.
stringsize	Returns control to the point at which the condition was detected.
error	Writes an error message on the 'error_output' stream and halts.
finish	Closes any open files and returns to the point at which the condition was detected.
(other language-defined conditions)	Writes an error message on the 'error_output' stream and signals the 'error' condition.
(programmer-defined conditions)	Writes an error message on the 'error_output' stream and halts.

'on' Condition Built-In Functions

Seven built-in functions are associated with the condition handling capability of PL/I. The 'on' condition built-in functions are used to access system variables whose values are set by the system when a condition is signalled. These functions allow the programmer to obtain information for use in the 'on' unit that handles the condition. Some of the 'on' condition built-in functions can be used as pseudovariables and, thus, the 'on' unit that handles the condition can also change some system variables.

Each 'on' condition built-in function is associated with a stack. When a condition that sets the value of an 'on' condition built-in function is signalled, the old value of the function is pushed down on the stack and the new value is placed on the stack. When control returns to the block activation in which the condition was signalled or to any of the dynamic predecessors of the signalling block, the value is removed from the stack. The stack provides for the possibility that the execution of an 'on' unit for a condition causes a condition to occur.

The 'on' condition built-in functions are given in the following list. For each function, the value of the system variable is given. Also listed are the names of all the conditions whose occurrence alter the stack.

<u>Built-In Function</u>	<u>System Variable Value</u>	<u>Associated Conditions</u>
onloc	name of the most currently entered procedure block	all
oncode	error code value	all
onchar	leftmost character for which the conversion failed	conversion
onfield	character string just extracted from the data stream	name
onfile	file name	conversion, name, endfile, transmit, record, key, endpage, undefinedfile
onkey	key value	endfile, transmit, record, key
onsource	string being converted	conversion

The occurrence of the 'endpage' condition, for example, provides a value for the 'on' condition built-in functions 'onloc', 'oncode', and 'onfile'.

If the condition is signalled by the execution of a 'signal' statement, the system variables have the values given in the following list:

<u>Built-in Function</u>	<u>System Variable Value</u>	<u>Associated Condition</u>
onchar	blank	conversion
onfield	null string	name
onfile	null string	conversion, name, endfile, transmit, record, key, endpage, undefinedfile
onkey	null string	key, endfile, transmit, record
onsource	null string	conversion

The 'onkey' built-in function is assigned a value by the 'signal' statement for the conditions 'endfile', 'transmit', and 'record' only if the file referenced has the keyed attribute.

A detailed description of the 'on' condition built-in functions is given earlier, in Section IX "Operations." The 'onchar' and 'onsource' functions can also be used as pseudo-variables. This use is described earlier, in Section X, "Value Assignment."

EXAMPLE OF CONDITION HANDLING

As an example of condition handling, consider the following program:

```
P:  proc;
    on conv
      begin;
        ... (print warning message)
        onsource() = "0";
      end;
    ...
    call Q;
    ...
Q:  proc;
    decl x float;
    call R(x);
    put skip;
    put list(x,sin(x));
    on conv
      begin;
        ... (print warning message)
        onsource() = "1";
      end;
    call R(x);
    revert conv;
    put list(x,1/x);
    call R(x);
    put skip list(x,x**2);
    end;
R:  proc(y);
    decl y float;
    get list(y);
    end;
end;
```

The program 'P' establishes an 'on' unit for the 'conversion' condition that prints a warning message and replaces the input string by the string "0". After some processing, the program 'P' calls 'Q'. The procedure 'Q' calls the procedure 'R' for input three times. The 'on' unit established by 'P' is suitable for the first and third call on 'R', but for the second call on 'R' a special 'on' unit is required.

The procedure 'R' represents a general-purpose input program, which would, in practice, be more complex. The procedure 'R' does not provide any handling for the 'conversion' condition because it cannot know the context of its call. The handling of the 'conversion' condition is entirely independent of the procedure in which it is signalled.

GUIDELINES FOR CONDITION HANDLING

The condition handling facility of PL/I is used both in debugging a program and in controlling the exceptional conditions that can occur during program execution. Guidelines for both applications are given here.

Debugging

During the debugging of a program, conditions that are normally disabled can be enabled and special 'on' units can be established.

ENABLING CONDITIONS FOR DEBUGGING

Four PL/I conditions provide additional error checking, namely; 'size', 'stringsize', 'stringrange', and 'subscriptrange'. These conditions are normally disabled, since their detection requires the generation of additional code in the object program to perform the testing. For debugging, these conditions should be enabled by preceding each external procedure with the prefix:

(size, strz, strg, subrg):

When the program goes into production, the prefix should be removed.

'on' UNITS FOR DEBUGGING

A useful debugging technique is the establishment of 'on' units to provide additional information about the state of the program when a condition is signalled.

Sometimes the same 'on' unit is established for every condition; namely, one that calls a debugging routine or produces standard information. Sometimes a different 'on' unit is established for each condition to produce debugging information specific to the condition. Sometimes, several 'on' units are established for the same condition, so the information produced depends upon the point at which the condition is signalled.

Controlling Exceptional Conditions

The condition handling facility of PL/I is used to control exceptional conditions. The file communication conditions are expected to occur and, consequently, 'on' units are usually established to handle these conditions. The error conditions, on the other hand, occur unexpectedly and the handling of these conditions is usually done by the default 'on' units provided by the system.

CONTROLLING FILE COMMUNICATION CONDITIONS

The 'on' unit established to handle a condition that communicates the status of an external file is part of the normal flow of a program. An example is a program that reads a file and uses an 'on' unit for the 'endfile' condition to transfer to the appropriate point in the program when the file is exhausted. A second example is a program that writes a report and uses an 'on' unit for the 'endpage' condition to write a footing and heading on the report. A final example is a program that accesses a keyed file and uses an 'on' unit for the 'key' condition to print a message and look in another file when the key is not found.

CONTROLLING ERROR CONDITIONS

Most programs do not establish 'on' units for the error conditions and if the condition is signalled, the default 'on' unit is executed. The default 'on' unit for most conditions prints an error message and terminates the execution of the program. In some cases, an alternative to the termination of the program can be defined. The following paragraphs consider these cases.

Input Data Validation

A program that reads input data often establishes an 'on' unit for the 'conversion' condition so that a bad input datum does not terminate the program. This 'on' unit can either report the bad input datum and read another or can attempt to correct the bad input.

Computational Checks

A program that is involved with computation often provides 'on' units for the 'overflow' and 'underflow' conditions to change the course of an algorithm so that processing can continue.

Resource Management

A program that includes a system of storage management based on areas often provides an 'on' unit for the 'area' condition.

Large Independent Systems

A large independent system or programming environment must handle all the language-defined conditions in order to maintain control over the processing. Moreover, such a system often makes use of programmer-defined conditions so that its users have the option of handling application-related conditions.

GENERAL CONDITIONS

The termination conditions are described in detail in this section since these conditions are related to the condition handling facility of PL/I and not to any particular PL/I language construct.

'error' Condition

The 'error' condition is signalled by the default 'on' units for several conditions, by the mathematical built-in functions, by the exponentiation operator, and by some run-time support routines.

The 'error' condition indicates a fatal error. If an 'on' unit for this condition attempts to return to the point at which the condition was signalled, the program is invalid and the results of its continued execution are undefined.

The default 'on' unit for the 'error' condition writes a comment on the 'error_output' stream and returns to command level. If the 'start' command is typed on the console after the default 'on' unit for an 'error' condition is executed, control returns to the point at which the condition was signalled. However, the program is invalid in this case.

'finish' Condition

In Multics PL/I, the 'finish' condition is signalled just before process termination.

The default 'on' unit for the 'finish' condition closes any open files and returns to the point at which the condition was signalled. If process termination resulted from the partial destruction of the process or exhaustion of process resources, the 'finish' condition is sometimes not signalled.

SECTION 14

STREAM INPUT/OUTPUT

In PL/I, the object from which input values are taken or to which output values are transmitted is called a data set. There are two kinds of data sets, stream and record, and PL/I has a complete input/output facility for each kind of data set. The facilities for stream input/output are oriented toward external media, such as line printers and card readers, and include elaborate features to assist the programmer in achieving a suitable format. In contrast, the facilities for record input/output are both more general and more primitive. This section describes the facilities for stream input/output, and the next section describes those for record input/output.

This section begins with a description of the two kinds of data that are involved in stream input/output: the stream, which is the actual subject of the input or output, and the file-state block, which shows the status of the operations on a stream. Next, the section describes the attachment of a PL/I data set to a Multics file. The section then gives a summary of the various operations that are performed as part of stream input/output. Once this foundation is established, the section proceeds to a definition of the statements that are used for stream input/output: first, the statements that open and close files, and then the statements that perform the actual input or output. Next, the section describes three different options for specifying the details of the format of stream input/output: data-directed, list-directed, and edit-directed. As the section nears its conclusion, a special feature of stream input/output, the string option, is presented. Finally, the section describes the conditions that occur in connection with stream input/output.

STREAM DATA SETS

The stream input/output statements operate on stream data sets or, more concisely, streams. A stream is a sequence of data characters and control characters. PL/I does not specify exactly what characters can be used as data characters, since this depends on the particular computer system and the input/output devices being used; however, any implementation of PL/I might be expected to have the letters, the digits, the common punctuation marks, and the blank among its data characters. There are two PL/I control characters, the linemark and the pagemark, and these represent the division between two lines or two pages, respectively.

A stream data set can be viewed as a character string that can be accessed only in special ways. During input, the characters of the stream are read in strict sequence, from left to right, and there is no way to return to a character that has already been read. During output, the characters of the stream are added at the right end of the stream, and there is no way to change a character that has already been written. The character string appears to flow past the PL/I interpreter and, for this reason, the data set is called a 'stream'.

The stream is a generalization of the various media that are used for communication between the user and the computer. It cannot handle diagrams or pictures, but it does capture the essence of the printed page or the punched card without introducing physical details of format, record boundaries, and so on. Furthermore, the processing of the stream is a good model for the behavior of hardware devices that are used for communication with the user. The single-pass, no-back-up property reflects the characteristics of line printers, card readers, and remote terminals.

The role of the stream is to make PL/I programs independent of the devices used for input and output. For example, a program can call for a sequence of one hundred arithmetic values without either telling a card reader to read another card or prompting an interactive terminal to supply another number. The program can call for input without knowing whether the input will come directly from a peripheral device, will be buffered, or will be waiting in permanent storage from some earlier input activity. Finally, by means that are discussed in this section, the program can deal with transmission errors and the end-of-file condition without reference to the specifics of the device that caused the condition to occur. Similar advantages apply to the process of stream output.

Control Characters in PL/I and Multics

PL/I assumes that the PL/I control characters, linemark and pagemark, are reserved for use in a stream and cannot be represented in a character-string value. In support of this assumption, PL/I provides statement options that are used to detect a linemark during input and generate a linemark or a pagemark during output. For example, to start a new page in the output stream, a program does not transmit a pagemark character; instead, it includes a 'page' option in the output statement, and the 'page' option causes a pagemark to be added to the output stream. This view of the control characters reflects the opinion that the starting of a new line or a new page is a rather special event in the composition of a printed document.

In Multics, the stream data set is a sequence of ASCII characters. The ASCII 'new line' and 'new page' characters are used for linemark and pagemark, and most of the remaining ASCII characters are used for the PL/I data characters.

A difference in principle exists between the design of the PL/I stream data set on one hand, and the Multics use of the ASCII character set on the other. PL/I views control characters as inseparably associated with the processes of input/output. Multics does not normally restrict any characters in this way and views input/output as just one of many operations to which a character-string is subject. Indeed, the definition of the PL/I character-string value, given in Section II, "Values," allows the use of any ASCII character in a character-string value. This difference in principle cannot be eliminated, but a practical solution can be achieved by asserting that

It is an error to use a stream input/output statement to attempt to transmit an ASCII carriage return, new line, backspace, tab, or new page character between the stream and PL/I storage.

This restriction leaves latitude for other uses of the restricted control characters; specifically, it allows for the inclusion of these control characters in the string values transmitted by record input/output statements.

Input Streams

When a pagemark occurs in the input stream, it is treated as a data character; thus an input stream is treated as a single page divided into any number of lines. When the stream data set is opened, a stream pointer is associated with it and set to the first character. As the file is processed, this stream pointer proceeds from one data character to the next, advancing by however many characters are read or skipped. When the stream pointer reaches the end of the stream, the 'endfile' condition occurs. *

For statements in which PL/I controls the editing of the input stream (that is, data-directed and list-directed input statements), the linemark has the effect of ending an item of input, and acts much as a blank does. This interpretation is in accord with the conventions of ordinary printed text, where an end-of-line can be used to separate two words.

For statements in which the programmer controls the editing of the input stream (that is, the edit-directed input statement), the linemark is ignored unless a specific reference to the line format is made. For example, if an input statement calls for three characters when the stream pointer selects the last character in a line, the resulting input will be the last character on the line and the first two characters on the next line; and no trace of the linemark will be input. On the other hand, an input statement can request that the stream pointer be moved to a certain column of a line or to the beginning of the next line.

There is no way to use stream input statements to program the operation "input the next line from the stream." When data-directed or list-directed input is used, the linemark is not distinguished from a blank. When edit-directed input is used, the programmer can skip the next line, but can perform input only by giving, in advance, the number of characters to be read. The operation in question can be performed by means of record input, as described under "Special Features" in this section.

Output Streams

When an output stream is opened, it is empty unless special arrangements are made to the contrary. As output is performed, characters are added at the right end of the stream. An output stream is usually intended for use to produce a printout. A 'print' output stream can contain both linemarks and pagemarks, and it thus exercises the full potential of the stream data set. A 'print' stream should (within the PL/I framework) be used only for printout; that is, the stream should not be used later as a PL/I input stream.

Generally speaking, PL/I allows a programmer to control the format of the output and intervenes only when the programmer neglects this activity. Specifically, the maximum length of a line and the maximum number of lines on a page are established when a stream data set is opened for output. If a program attempts to write beyond the end of a line, PL/I automatically inserts a linemark and forces the beginning of a new line. Similarly, if a program attempts to write beyond the end of a page, the 'endpage' condition occurs. The programmer can establish an 'on' unit for 'endpage' to start a new page and provide a suitable heading, or else he can allow PL/I to start a new page as the default response to the 'endpage' condition.

It is possible to use a stream data set as permanent storage. That is, a sequence of values can be written as output to a stream data set, left until they are needed, and then read as input from the same data set. However, this use of a stream data set is not recommended. It is difficult to imagine a case in which a record data set would not be a more efficient, simple, and accurate means for permanent storage.

Pseudo-Streams

A character-string storage unit can be used as if it were a stream, and in this role, it is called a pseudo-stream. For an input statement, characters are taken from the string variable just as if it were an input stream. For an output statement, the value of the string variable is first initialized to null and then characters are added to its value just as if it were an output stream. Under these circumstances, no actual input or output occurs, but the large and complicated editing facility of the stream input/output statements can be applied to the editing of character strings.

A pseudo-stream cannot contain either a linemark or a pagemark, and thus is viewed as a single line of data characters. This restriction is consistent with the PL/I requirements that control characters can only be used in (true) stream data sets.

Multics Files

The Multics implementation for a stream data set is an unstructured file. An unstructured file is a sequence of ASCII characters.

STREAM INPUT/OUTPUT FILES

A connection must be established between a statement that performs input/output and the Multics data set on which the operation is to be performed. A detailed analysis of this connection follows:

- o The connection begins with an input/output statement.
- o Every input/output statement has a file option, either explicitly given or obtained by default.
- o A file option has as its argument a file reference.
- o The evaluation of a file reference yields a file value.
- o A file value is a pointer-like object that designates a file-state block.
- o A file-state block is a structure-like set of values that includes a Multics data set designator.
- o A data set designator does, indeed, designate a Multics file and is thus the last part of the connection.

The PL/I data sets have already been discussed in this section, and the input/output statements will be discussed later. For the present, the subject is the file-state block and the file reference that designates it.

File-State Blocks

Transmission of values between PL/I and a Multics file requires bookkeeping data. This data is stored in a file-state block. When a file is open, the file-state block contains the designator of a Multics file and other information about input/output in progress. After the file is closed, the only information in the file-state block that is meaningful is that supplied by the attributes, if any, in the declaration of the file constant name. Although a file-state block resembles a structure, its values cannot be accessed directly by a PL/I program; instead, it is used and maintained by the PL/I processor.

The following values in the file-state block are relevant to stream input/output:

- The status indicator, which shows whether the file is 'open' or 'closed'; that is, whether or not a data set is currently attached to the file
- The Multics data set designator, which is used to establish the actual connection between the file-state block and a Multics file
- The file name, which is the file constant name (an identifier) expressed as a character-string value
- The file attributes associated with the current use of the file-state block. These are discussed later; one of them shows whether the file is an input file or an output file
- For an input file, the stream pointer, which points to the character which will be read next
- For an output file, the line size and page size, which give maximums for the number of characters per line and the number of lines per page, respectively
- For an output file, the column position, the line number, and the page number, which have values i, j, and k if the next character output will be the i+1-th character in the jth line on the kth page

It is customary to use the word "file" as an abbreviation for the term "file-state block", and some other liberties are taken to attain brevity. For example, one might say "advance the stream pointer associated with the file 'test2' through the input stream," instead of saying "advance the stream pointer contained in the file-state block associated with the file value designated by the file-constant 'test2' through the stream data set associated with the same file-state block." No confusion arises if one remembers that a remark about the actual data refers to a data set, while a remark about the control of the transmission process refers to a PL/I file-state block.

File References

A file-state block is designated by a file value; and the file value is supplied to an input/output statement by a file constant, a file variable, or a file-valued function. For example, the input statement:

```
get file(test2) list(a,b,c);
```

uses the file constant 'test2' to refer to a file-state block which, in turn, refers to a data set.

A file constant should be declared with the following attributes:

```
[external]
[internal] file [constant]
```

When the scope attribute is 'external', it can be omitted. The 'constant' attribute can always be omitted; however, some programmers prefer to write 'file constant' in a declaration in order to avoid confusion with a file variable.

Every file constant name must have an associated file description. It is recommended that this file description be given when the file is opened, as described later in this section, under "The 'open' Statement". However, PL/I does allow the programmer to write any portion of the file description attributes in the declaration of the file constant name.

For each declaration of a file constant, a file-state block exists in static storage. The only exception is the declaration of a given identifier as 'external file constant' in several places; in this case, the declarations refer to the same file-state block, as required by the proper interpretation of the 'external' attribute. A given file constant and its associated file-state block can be used for more than one data set in the course of a PL/I program execution. For example, a file-state block can be opened for input from a stream data set, closed, opened for output to a record data set, and closed again.

A file variable or file function is declared similarly to an arithmetic variable or function. The only differences are:

- The data type is 'file'.
- The default scope 'external' applies to a file variable, whereas the default scope for most other data types is 'internal'.
- The attribute 'variable' must be used explicitly for a file variable because PL/I will otherwise assume the identifier is a 'file constant'.

FILE 'ATTACHMENT

The Multics I/O system uses a software construct, the I/O switch, to control the source or destination of an input/output operation. A PL/I file-state block is attached to a Multics file through a named switch. The switch name and the file name are the same for an external file. For an internal file, a unique name is generated for the switch.

Two operations, attachment and opening, are associated with I/O switches. When an I/O switch is attached, the source or target and the I/O module that performs the input/output are established. When an I/O switch is opened, a particular mode of processing is established.

Attaching a Switch

An I/O switch can be attached either at command level by the 'io_call' command or within a PL/I program by the execution of an input/output statement. If the switch is not attached when the 'open' statement for the associated file is executed, the information in the 'title' option (given explicitly or by default) is used to attach the switch. An I/O switch attached by the execution of an 'open' statement for a file is detached when the 'close' statement for the file is executed. If an I/O switch is already attached, neither the 'open' nor the corresponding 'close' statement has any effect on the switch's attachment.

ATTACH DESCRIPTION

The 'title' option contains the attach description. The attach description specifies an I/O module to perform the input or output operation and the file or device to be used as the source or destination for these operations. For stream input/output the following I/O modules can be used:

<u>I/O Module</u>	<u>Usage</u>
vfile_	for storage-resident files
tty_	for terminal attachment
syn_	for synonym attachment
record_stream_	for conversion between stream and record files

The form of the attach description depends upon the I/O module. Each of the above I/O modules is described later, in Section XVI, "PL/I in the Multics System." For an example of an attach description, consider the following 'open' statement:

```
open file(str1) title("vfile_ alpha") input;
```

The attach description in the 'title' option specifies that the I/O module 'vfile_' is to be used to perform input from the system-resident file in the segment 'alpha'.

If no 'title' attribute is given, a default attach description is formed, as follows:

```
syn_ user_input          (for 'sysin')
syn_ user_output        (for 'sysprint')
vfile_ fn                (for file fn)
```

where fn is the file name other than 'sysin' or 'sysprint'.

Opening a Switch

An I/O switch can be opened either at command level by the 'io_call' command or within a PL/I program by the execution of an input/output statement that references the file associated with the switch. If the switch is not open when the 'open' statement is executed for the file, the information in the file description is used to open the switch. If the file description is omitted or is incomplete, a default assumption is used. If the file is opened by an input/output statement other than the 'open' statement, the file description is derived from the type of statement. The file description attributes specify the opening mode of the switch.

Opening a File

A PL/I file-state block, or file, is opened by the execution of the first input/output statement referencing the file. The file name, title, and file description are passed to the Multics I/O System to open the file. If any of these options is not explicitly given, a default assumption is derived from the input/output statement.

The Multics I/O system uses the title to attach the switch if it is not already attached and the file description to open the switch if it is not already open. Then the Multics I/O system returns a data set designator. This data set designator makes the connection between the PL/I data set and the Multics file. The data set designator is stored in the file-state block for use when an input/output operation is performed on that file.

STREAM INPUT/OUTPUT OPERATIONS

The summary of stream input/output operations that follows includes terminology, shows how stream data sets are manipulated, and gives a general view of the stream input/output facility. It also serves as an introduction for the subsequent pages of this section.

When a file is opened for input the designated Multics file is attached and preparations are made for reading from the first character onward. When a file is opened for output, the previous contents of the designated Multics file are discarded and the file is prepared for the appropriate output format. A stream file can be opened for input or for output, but not for a combination of input and output.

Input operations proceed through the file reading or skipping characters in strict sequential order. Output operations write characters, always adding them at the right end of the stream. Thus the actual transmission of data is simple. The complexity of stream input/output arises from the variety of ways in which the 'get' statement edits the characters that are read from the input stream and the equally numerous ways in which the 'put' statement edits the characters that are added to the output stream.

Within stream input/output, there are three separate disciplines of input/output. The first two disciplines, data-directed and list-directed, are closely related and are both quite automatic; that is, PL/I makes most of the decisions about the representation of values and the layout of a page. The third discipline, edit-directed, allows the programmer to specify the details of representation and layout. Within this last discipline there is yet a further choice of methods; the programmer can choose between format items, which are derived from FORTRAN, or pictures, which are derived from COBOL. Clearly, the choice of an input/output discipline can be difficult, and advice on the choice will be given.

Many conditions can occur as the direct or indirect result of stream input/output. Certain conditions always imply that an error has occurred; other conditions are used to control the logic of the input/output process and do not necessarily indicate an error. Some conditions are uniquely associated with input/output; other conditions arise from common operations, such as the assignment of a value to a variable, which happen to be used during input/output.

This concludes the summary of stream input/output operations. The remainder of this section is devoted to the detailed consideration of these operations.

OPENING AND CLOSING FILES

When a file is opened, the file-state block is marked "open" and the data set designator, control parameters, and indexes are set in the block. When a file is closed, the file is marked "closed" and only information provided by the file declaration is meaningful.

A file is opened when the first input/output statement referencing the file is executed. The purpose of the 'open' statement is to provide the title and file description for the file opening. However, both these options can be omitted from the 'open' statement, and, in that case, a default assumption is made. If an 'open' statement is not given for a file, the attributes for the file opening are derived from the first input/output statement executed. If a file is already open when an 'open' statement is executed, the 'open' statement is completely ignored.

'open' Statement

When an 'open' statement is used to open a stream for input, it gives a file value, a title for the stream data set, and a file description. Consider the statement:

```
open file(test2) title("vfile_ alpha>beta>gamma") input;
```

In this statement, the file value is given by the constant 'test2', the title is the Multics attach description 'vfile_ alpha>beta>gamma', and the file description is 'input'. This statement is interpreted as follows:

- The title is used to provide the attach description for the I/O switch if the switch is not already attached.
- The associated Multics file is checked to make sure that it does not have any attributes that conflict with 'stream' and 'input'.
- The column position associated with the file is set to '0', and the stream pointer is set to point to the first character of the stream file.
- The file is marked 'open'.

If any of these steps cannot be performed, then the 'undefinedfile' condition occurs. If the file designated by 'test2' is already open when the statement is executed, then the 'open' statement is ignored.

When an 'open' statement is used to open a stream for print output, the maximums for the length of a line and a page can also be given. Consider the statement:

```
open file(report) title("vfile_ alpha>beta>gamma") print output  
linesize(80) pagesize(50);
```

The statement is interpreted as follows:

- The property page size associated with the file is set to the value given by the pagesize option, namely 50.
- The property line size is set to the value given by the linesize option, namely 80.
- The title is used to provide an attach description as in the previous 'open' statement.
- The data set is deleted and a new data set conforming to the file description 'stream print output' is created.
- The indexes line number and page number are set to '1' and the index column position is set to '0'.
- The file is marked 'open'.

As in the previous example, the 'undefinedfile(report)' condition occurs if any of these steps cannot be performed, and the 'open' statement is ignored if 'report' is already open.

If the title option is omitted from an 'open' statement, the file name is used in forming a title. The statement

```
open file(report) print;
```

is equivalent to

```
open file(report) title("vfile_report") stream print output  
linesize(132) pagesize(60);
```

This example also shows that the default maximum for the length of a line is 132 characters, the default maximum for the number of lines on a page is 60, and 'print' implies an output stream.

There are not many file descriptions for stream input/output. The important ones were used in the example of the 'open' statements, above; they are

$$\left\{ \begin{array}{l} \text{input} \\ \text{print } [\text{output}] \end{array} \right\}$$

The 'print' attribute should not be used in opening an output stream that, at a later time, will be used as an input stream. When an output stream is directed at an interactive terminal, the 'environment(interactive)' attribute should be used. Altogether, the file descriptions for a stream data set are:

$$\left\{ \begin{array}{l} \text{input} \\ \text{output} \\ \text{print } [\text{output}] \\ \text{output environment } [(\text{interactive})] \\ \text{print } [\text{output}] \text{ environment } (\text{interactive}) \end{array} \right\}$$

When the 'environment(interactive)' attribute is used, each 'put' statement that operates on the file adds a linemark to the end of its normal output.

'close' Statement

The 'close' statement has a simple form, as indicated by the following example:

```
close file(test2);
```

This statement marks the file-state block 'test2' closed. If the program is interrupted before a file has been closed, its contents are undefined. Therefore, every 'open' statement should be matched by a 'close' statement; and, further, the 'close' statement should be executed as soon as possible after the completion of input/output operations on the file.

Default Files

It is possible to write a PL/I program without declaring a single file constant or using a single 'open' statement. In that case, input is taken from the standard input stream, 'user_input', and output goes to the standard output stream, 'user_output'. Consider first the case of a stream input statement that does not have a file option; for example,

```
get list(a,b,c);
```

The default mechanism of PL/I comes into play with full force here. Before execution of the program begins, PL/I inserts an 'open' statement and adds a file option to the 'get' statement, giving

```
open file(sysin) title("syn_ user_input") input;  
get file(sysin) list(a,b,c);
```

If the statement does not lie within the scope of a declaration of 'sysin', PL/I supplies the following declaration in the largest enclosing block:

```
dcl sysin external file constant;
```

The omission of an 'open' statement is common. On the other hand, 'sysin' and 'sysprint' should always be declared because the Multics PL/I compiler gives a warning message for any undeclared name.

The default interpretation just described is applied to every 'get' statement in a program that does not have a 'file' option. Because 'sysin' is declared 'external' all such statements refer to the same file-state block, even when the interpretation of several 'get' statements leads to several declarations of 'sysin'. Because an 'open' statement only performs an action when the designated file is not already open, the file 'sysin' is only opened once.

Consider next a stream output statement that does not have a file option; for example,

```
put list(x,y,z);
```

This statement is treated like the input statement, except that 'sysprint' is used as the default file constant. The statement is expanded to be:

```
open file(sysprint) title("syn_ user_output") print;  
put file(sysprint) list(x,y,z);
```

INPUT/OUTPUT STATEMENTS

There is one statement for stream input, the 'get' statement, and one statement for stream output, the 'put' statement. Each statement is a keyword followed by a sequence of options. Most of the options are simple. They specify the file on which the statement will operate, the number of lines to be skipped before reading or writing begins, and so on; and these simple options are described here. The final option in a 'get' or 'put' statement is the transmission option and it is not simple. It specifies the transmission of data according to the rules of data-directed, list-directed, or edit-directed input/output. The transmission option is described later in this section.

In addition to the 'get' and 'put' statements, there is a third stream input/output statement, the 'format' statement. This statement plays a specialized role, and is used only in connection with an edit-directed transmission option.

'get' Statement

Stream input is performed by the 'get' statement. A full example of such a statement follows:

```
get file(test2) copy(echo9) skip(n+2) list(a,b,c);
```

There are four options in this example. The options are interpreted as follows:

- 'file(test2)' is the file option. It designates a file-state block and, through the file-state block, the stream data set from which input is to be taken. If the option is omitted, 'file(sysin)' is assumed.
- 'copy(echo9)' is the copy option. When this option appears, every character skipped or read from the input stream is written in the output stream designated by this option. If a 'copy' is given without an argument, 'copy(sysprint)' is assumed.
- 'skip(n+2)' is the skip option. It specifies that, before any input is performed, characters will be skipped until the beginning of the (n+2)th line after the current line. If 'skip' is used without a count, 'skip(1)' is assumed.
- 'list(a,b,c)' is the transmission option. In this case it is list-directed and specifies that the next three values in the stream will be read in and assigned to the three targets, 'a,b,c' given in the list. If the option is omitted, no values are input.

The example just given shows the important input options. The following points round out the description of the 'get' statement:

- The skip and transmission options cannot both be omitted, since the statement would then do nothing; but options can be omitted in any other way. In fact, the use of all options, as in this example, is rare.

- The options can be arranged in any order, but the order used in this example is recommended because it is the order in which the actions are performed. The source and copy files are prepared for transmission, then the skip is performed, and only then does input from the stream occur.
- It is possible to simulate input by using the option `'string(e)'` instead of the file option, where e is a character-string expression. The value of e is treated just as if it were the current input stream. More is said of this later, under "The String Option".

'put' Statement

Stream output is performed by the `'put'` statement. A full example of such a statement follows:

```
put file(report) page line(5) list(x,y,z);
```

There are four options in this example. The options are interpreted as follows:

- `'file(report)'` is the file option. It designates the stream data set to which the output will be transmitted. If the option is omitted, `'file(sysprint)'` is assumed.
- `'page'` specifies that a pagemark will be written in the output stream.
- `'line(5)'` specifies that sufficient linemarks will be written so that subsequent output will begin the fifth line of a page. If a new page must be started, the subsequent output begins on the first line of the new page.
- `'list(x,y,z)'` is the transmission option. In this case it is list-directed and specifies the values that are to be written out. If the option is omitted, no values are output.

The example just given does not illustrate the following features of the `'put'` statement:

- A `'skip(n)'` option can be used in a `'put'` statement. It specifies that subsequent output should begin the nth line after the current line. If `'skip'` is used without an argument, `'skip(1)'` is assumed. If the skip option is used, neither the page nor line option can be used.
- The skip, page, line, and transmission options cannot all be omitted, since the statement would do nothing; but options can be omitted in any other way.
- The options can be arranged in any order, but the order used in the example is recommended because it is the order in which actions are performed: The destination file is located, then the page and line options are performed (in that order), and only then does output to the stream occur. If a skip option is used, it should be written just after the file option so that it takes the place of any page or line option.
- It is possible to simulate output by using `'string(t)'` where t is a suitable target for assignment of a character string value. The string of characters that would otherwise be placed in the output stream is assigned to t.

'format' Statement

The 'format' statement is the keyword 'format' followed by a parenthesized format list. The statement must begin with at least one label prefix. Consider the statement

```
F5: format(a(10),p"bbb--9v.99");
```

This statement has the label prefix 'F5:' and the format list is made up of the items 'a(10)' and 'p"bbb--9v.99"'.

The purpose of a format statement is to supply a format list for an edit-directed 'get' or 'put' statement that appears elsewhere in the program. Specifically, such a 'get' or 'put' statement may contain a remote format item, such as

```
r(F5)
```

The remote format item is interpreted by interpreting the items in the format list associated with 'F5'; namely,

```
a(10), p"bbb--9v.99"
```

Thus the 'format' statement is used in order to supply a format list for some other stream input/output statement. The interpretation of the format list itself will be given later, in the discussion of edit-directed input/output.

A 'format' statement is similar in some respects to a procedure. The identifier 'F5' is a format constant name, not a label constant name. It is used to invoke the 'format' statement only in a remote format item, and it cannot be used as the destination of a transfer of control. When control reaches a format statement as a result of the sequential execution of the preceding statements, the format statement is skipped, just as a procedure is skipped under the same circumstances.

The value of a format constant is similar in many respects to an entry value. The format value can be assigned to a format variable, can be the result of a reference to a format variable or a function, and can be compared to another format value by means of the operators '=' and '^='. A variable or function that has a format value is declared similarly to an arithmetic variable or function; its data type is 'format', and the attribute 'variable' is always assumed to apply.

As an example of the use of a format variable, and as a further example of the format statement, consider the following program fragment:

```
...
dcl X format;
Q:  format(a(10),f(11,2));
...
X = Q;
...
get edit(x, y)(r(X));
...

```

The remote format item has the format variable 'X' as its argument. Since the value of 'X' is the format constant 'Q', the remote format item is equivalent to 'r(Q)'. Furthermore, since the format constant 'Q' is associated with the format list '(a(10), f(11,2))', the 'get' statement is equivalent to

```
get edit(x, y)(a(10), f(11,2));
```

The interpretation for this kind of input statement is given later; the purpose here has been to show how format constants, variables, and statements can be used to supply a format list to such a statement.

DATA-DIRECTED INPUT/OUTPUT

When data-directed input is used, the input stream contains a variable name for each input value; so the target for an input value is provided by the data rather than by the program. Furthermore, once a data-directed statement has initiated input, the process continues until a semicolon is encountered in the input stream; so input is terminated by the data rather than the program. For these reasons, the term "data-directed" is applied to this input/output facility.

Examples of Data-Directed Input/Output

As an example of data-directed input/output, consider the following program to calculate the range of an artillery piece fired on level ground:

```
RANGE:  proc;
        dcl (sysin,sysprint) file;
        dcl (v0, theta, range) float(15),
        dcl g float(15) init(32.174);
        do while("1"b);
            get data(v0, theta);
            if v0=0 then return;
            range = ((v0**2)*sind(2*theta))/g;
            put skip data(v0, theta, range);
            put skips;
            end;
        end;
```

This program appears to be an infinite loop (since the condition 'while("1"b)' is always satisfied). Each time around the loop, the program reads the muzzle velocity (v0) and the angle of elevation (theta), performs an end test, calculates the distance to impact (range), and outputs results. The program is designed to stop on a misfire; that is, it returns when v0=0.

The input statement in this program is the data-directed 'get' statement

```
get data(v0,theta);
```

Each time this statement is executed, the input stream is read through the next semicolon in the stream. Suppose the program is being used to calculate four trajectories. Then the input stream might be:

```
v0=1000  theta=35;
v0=1000  theta=40;
v0=1000  theta=45;
v0=1280  theta=45;
v0=0     theta=45;
```

In this example, each line ends with a semicolon and therefore represents the two values required for an execution of the 'get' statement and the calculation of a trajectory. The zero value of 'v0' stops the program, but the last value of 'theta' is added for completeness and has no effect whatever.

A value not mentioned in the input stream is not changed; furthermore, the order in which values are mentioned has no significance. Therefore, the same trajectories can be specified as follows:

```
theta=35      v0=1000;
theta=40;
theta=45;
v0=1280;
v0=0;
```

The stream assignments can be separated from one another by any sequence of blanks, tab characters, and new lines; and thus the input data can be attractively formatted (as above) and can be broken into several lines when it does not fit on one line.

It is the semicolon and not any other character that delimits the input stream read by a given 'get' statement. Thus the example can be written in yet another way, this time as a compact single line:

```
theta=35 v0=1000;theta=40;theta=45;v0=1280;v0=0;
```

This format is less attractive but, as a practical matter, it might well be used when the input is being typed in at an interactive terminal.

Why are the variables 'v0' and 'theta' mentioned in the 'get' statement? Since the input stream specifies a variable for each input value, a mention of variables in the 'get' statement appears to be redundant -- and it is. The effect of the mention of variables in the 'get data' statement is to restrict the input stream to those variables only. Thus, for example, if the input stream given above included 'g=1000', PL/I would reject the input and cause the 'name' condition to occur because the 'get data' statement does not mention 'g'. This restriction allows the programmer to maintain control over the effects of data-directed input and also allows PL/I to execute data-directed input more efficiently.

The output statement in the RANGE program is the data-directed 'put' statement

```
put skip data(v0, theta, range);
```

If the input stream is the one discussed above, the program executes this 'put' statement four times and produces the following addition to the output stream:

```
v0= 1.0000e+003      theta= 3.5000e+001      range= 2.9207e+004;
v0= 1.0000e+003      theta= 4.0000e+001      range= 3.0609e+004;
v0= 1.0000e+003      theta= 4.5000e+001      range= 3.1081e+004;
v0= 1.2800e+003      theta= 4.5000e+001      range= 5.0923e+004;
```

The preceding output is directed to the 'sysprint' file, which is declared 'print'. For a file with the 'print' attribute, PL/I assumes the tab stops of the printer are set at 11, 21, 31, and so on. Each stream assignment (except the first) is preceded by a tab character, and each stream assignment is followed by a blank. In the example above, the three stream assignments are 15, 18, and 18 characters in length so they begin in columns 1, 21, and 41.

When a value is output, it is first converted to a character string and then output in that form; this conversion is discussed in Section IV, "Value Conversion." In the RANGE program, all of the values have the same data type, namely 'float(15)'. The conversion of such a value to a character string proceeds as follows: the value is converted from 'binary' to 'decimal' with an equivalent precision; in this case, the target data type is 'dec float(5)'. Then the decimal value is converted to a character string in a straightforward way to give a string of 12 characters.

In a second example, the following program illustrates the remarkable flexibility of data-directed input:

```
UPD: proc;
      dcl (sysin,sysprint) file;
      dcl K char(10);
      dcl 01 M,
          02 name char(30) var,
          02 address,
          03 street char(30) var,
          03 CSZ,
          04 city char(20) var,
          04 state char(2),
          04 zip pic"99999",
          02 expire,
          03 month pic"99",
          03 year pic"99",
          02 account pic"$$$9.99";
      dcl members file;
      open file(members) keyed update;
      do while("1"b);
        get data(K);
        if K="" then do;
          close file(members);
          return;
        end;
        read file(members) key(K) into(M);
        put data(M);
        get data(M);
        rewrite file(members) key(K) from(M);
      end;
end;
```

This program uses record input/output and therefore anticipates the reader's progress through the manual; however, the program is easily explained.

The program assumes that a record data set exists that describes the membership of an organization, giving the status of annual dues for each member. Each record is accessed by means of a character string, the key, and has a value that can be stored in the structure 'M' in the program. The main part of the program is a loop. Each time through the loop, the program gets a key from the user, stops if the key is an empty character string, uses the key to read the value of a record from the file into the structure 'M', prints a copy of the value, gets modifications of the value from the user, and writes the modified value on the record.

The interesting part of the program is the statement 'get data(M);'. This statement allows the user to enter modifications to any components of (M). For example, the input stream might be as follows:

```
K= "CRIEM06301";
month= 3   year= 74;
K= "MAREM06733";
zip= 02139;
```

Two records are changed. For the member whose key is 'CRIEM06301', the date of membership expiration is changed to March, 1974. For the member whose key is 'MAREM06733', the zip code is changed to read '02139'.

Principles and Exceptions

The underlying principle of data-directed input is as follows: when a stream assignment is read by a 'get data' assignment, it is treated as if it were an assignment statement that appeared exactly where the 'get data' statement appears. For practical reasons, the following exceptions apply:

- The assignment must not require computation. If the target variable has subscripts, they must be signed or unsigned integers. The value on the right must be a value that could appear in a storage unit just as it is; for example, '4+5' is not allowed, but '4+5i' is.
- The variable name must designate a scalar storage unit. The variable name must appear explicitly in the 'data' list or else it must designate a component of an aggregate variable whose name appears in the 'data' list.
- Stream assignments are not separated by semicolons (except where semicolon is used to terminate input). Instead, a sequence of blanks, tab characters, linemarks, and commas can be used. Blanks, tabs, and linemarks can also be used adjacent to the '=' sign.

The underlying principle of data-directed output is as follows: if a 'put data' is executed for a given list of variables and produces a certain output stream, then the execution of a 'get data' with the same list of variables on an identical input stream will assign to the variables their original values. It is not suggested that one would actually perform this operation; it is a way to store values in a stream, but far inferior to the facilities of record input/output. However, this principle does provide an understanding of the design of data-directed output. For example, a 'put data' statement adds a semicolon to the end of its stream output so the stream would be delimited if it were used for data-directed input. Exceptions to this principle of reversibility are:

- If the output file has the 'print' attribute, the quote marks are removed from the value of a character string. This allows character values to be used for identification of output but, at the same time, interferes with their being read back as input character strings.
- For all output files, a 'binary' arithmetic value is converted to base 'decimal'. This may result in the loss of some precision, so that when the value is converted back during input it will not be quite the same 'binary' arithmetic value.

Guidelines for Data-Directed Input/Output

Data-directed input/output is best suited for temporary applications; either in a program written quickly and used only a few times, or as temporary diagnostic output in a program being tested. The latter application of data-directed input/output is a useful debugging aid. Appropriate 'put' statements can be inserted to produce dumps of specific data without regard to the format of the output. If each such statement is marked with a comment indicating its role, such as '/*dump*/', the statements can be systematically removed when debugging is complete.

Data-directed input/output is the most device-independent form of input/output. PL/I arranges stream assignments in a way that is readable and that is usually neatly aligned in columns. Since each value is paired with an identifying variable, no reasonable arrangement of the data can interfere with the proper identification of the values.

Data-directed input/output is well protected against user errors. If the user places a variable in the input stream that does not appear in the 'data' option, the 'name' condition occurs; and an array subscript that is out of bounds is similarly treated. There is a fairly good chance that a value will either arrive at its intended variable or will be rejected as invalid.

These advantages are balanced by disadvantages. Data-directed input/output is the least efficient of the modes of input/output since the assignment of input data must be determined entirely at execution time. The preparation of input data requires more keystrokes, since each value must be preceded by its variable name. Output data can become cluttered by the repeated occurrences of just a few variables. For all of these reasons, data-directed input/output is usually more appropriate for small-scale, simple transmission of data.

LIST-DIRECTED INPUT/OUTPUT

When list-directed input is used, the variable that is the target of an input value is given in a list that is part of the 'get' statement; so this list determines where the input values go. For this reason, the term "list-directed" is applied to this input/output facility.

Example of List-Directed Input/Output

The program to calculate the range of an artillery piece is used here as an example of list-directed input/output. That program was given under "Data-Directed Input/Output", and only the two input/output statements need be changed, as follows:

```
RANGE2: ...
        ...
        get list(v0, theta);
        ...
        put skip list(v0, theta, range);
        ...
        end;
```

List-directed input/output is quite similar to data-directed input/output, and the program above is testimony to this fact. RANGE2 differs from RANGE only in the use of the keyword 'list' instead of 'data' in the input/output statements. Suppose this program is being used to calculate four trajectories, just as RANGE was. Then the input stream would be:

```
1000  35
1000  40
1000  45
1280  45
0     45
```

This input is the bare minimum and looks more like computer input prepared in bulk, on punched cards, for example. The input is nicely formatted, but PL/I is not influenced by that. If, by mistake, the stream pointer is just after the '1000' on the first line, the whole run of the program will be invalid; indeed, it will not stop until a zero is encountered somewhere beyond the portion of the stream shown above or until some input/output condition stops it.

It is possible to ignore a target in the 'get' statement by leaving a value blank. In order to do so, however, commas must be used to separate the values, as follows:

```
1000,  35,
      ,  40,
      ,  45,
1280,  ,
      0,  ,
```

Again, the format is for the benefit of the user, not PL/I. The stream could be entered as

```
1000,35,,40,,45,1280,,0,,
```

with exactly the same effect. The use of a comma as a separator is optional in list-directed input, but it is suggested that commas be used throughout when one or more targets are ignored.

If the input stream is the one just discussed, the program produces the following addition to the output stream:

1.0000e+003	3.5000e+001	2.9207e+004
1.0000e+003	4.0000e+001	3.0609e+004
1.0000e+003	4.5000e+001	3.1081e+004
1.2800e+003	4.5000e+001	5.0923e+004

This output will be labeled if the following statement is inserted before the loop:

```
put skip list("velocity", "elevation", "range");
```

Note that each heading is padded with blanks so it is just as wide as the value it labels; this assures that it will line up whatever tab stops are used. The blank at the beginning of each heading corresponds to the sign position of the numbers. The statement prints the line:

```
velocity          elevation          range
```

at the beginning of the output.

Compound List Items

In its simplest case, a 'get list' statement has a list of designators of scalar values, and these are paired, one-for-one, with the values in the input stream. The 'get list' statement in the RANGE2 program was of this sort: its list was composed of two scalar names, 'v0' and 'theta'. However, list-directed input is not restricted to this simple case; instead, an item can be any variable name, scalar or aggregate, or it can be a parenthesized iterated list of items. Such items are interpreted by expanding them from a single item into a sequence of items; and they are therefore called compound items.

An item that is an array variable name represents the elements of the array listed in row-major order. Suppose the following declaration is in effect:

```
dcl A(3:4,3) float;
```

Then the statement

```
get list(A);
```

is interpreted as

```
get list(A(3,1), A(3,2), A(3,3), A(4,1), A(4,2), A(4,3));
```

and therefore reads six values from the input stream.

An item that is a structure variable name represents the members of the structure listed in the order in which they are declared. If a member of the structure is not a scalar, then the member is expanded, in turn, into a list of its components. Suppose the following declaration is in effect:

```
    decl 01 specs,  
          02 side(2),  
            03 h float,  
            03 w float,  
          02 area float;
```

Then the statement:

```
    get list(specs);
```

is interpreted as:

```
    get list(specs.side(1).h, specs.side(1).w,  
            specs.side(2).h, specs.side(2).w, specs.area);
```

and therefore reads five values from the input stream. The item in this example is a "level one" structure; but any component of an aggregate can be referred to. For example, 'get list(specs.side)' is interpreted as above but with 'specs.area' omitted.

An item that is a parenthesized iterated list represents a list of subscripted items. For example, the statement:

```
    get list((x(i), y(i) do i = 2 to 4));
```

is interpreted as:

```
    get list(x(2), y(2), x(3), y(3), x(4), y(4));
```

and therefore reads six values from the input stream. The form of the iterated list is based on the 'do' statement, and can use any of the multiple do clauses defined in Section XI, "Program Flow." The following example illustrates the options allowed:

```
    get list((busy(k) do k = 1, 5, 6 repeat 2*k while(k<25),  
            23 to 15 by -2));
```

This statement uses the following sequence of subscripts:

```
    1, 5, 6, 12, 24, 23, 21, 19, 17, 15
```

Therefore it reads ten values from the input stream into the array 'busy'.

All of the compound items have now been informally described; but without further examples, certain useful techniques might be overlooked by the reader. For example, an item in a parenthesized iterated list can itself be a compound item. The statement:

```
    get list((Q(i,*) do i = 1 to 3));
```

reads in values for the first three rows of the array 'Q'. In this example, 'Q(i,*)' is a compound item representing the values in a row of the array 'Q'.

An item in a parenthesized iterated list can be another parenthesized iterated list, as shown in the following statement:

```
get list(X, ((A(j,k) do k = 1 to 2), B(j) do j = 3 to 4), Y);
```

This statement is interpreted as:

```
get list(X, A(3,1), A(3,2), B(3), A(4,1), A(4,2), B(4), Y);
```

and therefore reads eight values from the input stream.

Any appropriate expressions can be used in the clauses of the multiple do. For example, the statement:

```
get list(x, (y(m) do m = m1 to rad*u(j-1)));
```

can be used, but of course its interpretation cannot be determined until, as the result of the execution of earlier statements, values have been assigned to 'm1', 'rad', 'j', and 'u(j-1)'. Even values previously input by the same 'get' statement can be used to control the statement; for example, the statement:

```
get list(n, (w(i) do i = 1 to n));
```

reads a value into 'n' from the input stream and then uses that value to determine how many values are read into the array 'w'.

The list of a 'put list' statement can use the compound items that have been described for the 'get list' statement. Each compound item in a 'put list' statement is interpreted as a sequence of items in exactly the way it would be interpreted in a 'get list' statement.

Each output value in a 'put list' statement can be given by an expression of unlimited complexity. Just as a 'get list' allows the use of anything that could appear on the left side of an assignment statement, so a 'put' statement allows use of anything that could appear on the right side of an assignment statement.

Pseudo-Variable List Items

Since an item in a 'get list' statement can be any target, it can be a pseudo variable. For example, the statement

```
get list(real(X),imag(X));
```

reads two values from the input stream and assigns them as the real and imaginary parts of 'X', respectively. The values must be 'real' and the variable 'X' must be 'complex'.

Principles and Exceptions

The principle of list-directed input is as follows: the variable names in the list in the 'get' statement are processed from left to right; and, for each variable, an assignment statement is executed that consists of the variable name, an '=' sign, and the next item from the input stream. In practice, list-directed input is both more powerful and less powerful than an ordinary assignment statement, as indicated by the following exceptions:

- When a variable designates an aggregate, a sufficient number of (scalar) values are taken from the input stream to provide a complete value for the variable, as described under "Compound Items", above. There is no corresponding feature of an ordinary assignment statement because there is no way to write an aggregate constant. For example, one cannot write 'A = 89,-99,17;' to assign an array value to 'A'.
- Items read from the data stream must be values as they stand, without further computation.
- The reference that appears in a 'get list' list must have an arithmetic or string data type since only values of these types can appear in the input stream.

The principle of list-directed output is the same as that for data-directed output; whatever is output by a given statement can be input by a statement with the same list. As with data-directed input/output, the exceptions to this principle occur when character strings are output under the 'print' attribute and when a 'binary' value is converted to 'decimal' for output.

Guidelines for List-Directed Input/Output

List-directed input/output is more efficient than either data-directed or edit-directed input/output because it does not require the execution-time interpretation of variable names (as does data-directed input/output) or execution-time editing (as does edit-directed input/output). On grounds of efficiency alone, then, it is preferred; but its limited and rigid format is a disadvantage, especially for output.

For list-directed output, the programmer cannot choose a format freely; he must take into account the tab stops provided on the printer, for example. Further, automatic conversion of arithmetic values to character strings uses moderately complicated rules and can produce some surprises. Thus, it is often easier to use edit-directed output from the outset and be assured of full control over the output format.

The use of list-directed input is more attractive, since the insensitivity to format is an advantage. If a large volume of input is required, it can be typed or punched value after value, line after line, using exactly the precision and scaling that appear in the raw data.

EDIT-DIRECTED INPUT/OUTPUT

When edit-directed input/output is used, the statement includes instructions for the editing of the transmitted values. The edit-directed statement includes a list of data items, just as a list-directed statement does, but also has a list of format items that control the editing of each value, either checking the format of input or supplying the format for output. For this reason, the term "edit-directed" is applied to this input/output facility.

Examples of Edit-Directed Input/Output

Edit-directed input/output will be introduced through discussion of several versions of a single program. The program inputs a number, divides it by two, and outputs both the given number and the computed result. It is known that the input is supplied as a signed, three-digit number. The first version of the program is:

```
P1:  proc;
      dcl (sysin,sysprint) file;
      dcl (x,y) float;
      get edit(x)(a(4));
      y = x/2;
      put edit(x,y)(skip,a,x(3),a);
      end;
```

This program uses the most fundamental of all format items, the 'a' (for "alphanumeric") format item. This format item specifies the reading of characters from the input stream without any checking or conversion whatever.

The 'get' statement is interpreted as "read the next four characters from the input stream and assign them (as a character-string value) to 'x'". It performs a very simple input operation. A conversion of data type does occur, but it is part of the assignment of the character string input to 'x', and it is not controlled by the 'get' statement. Suppose the input stream currently begins with '-709'; then '-709' is expressed as a 'real binary float(27)' value and assigned to 'x'. The computation part of the program assigns '-354.5' to 'y'.

The 'put' statement is interpreted as "skip to the beginning of a new line; output the value of 'x' as a character string; output three blanks; and output 'y' as a character string". Since the values of 'x' and 'y' are not character strings, these values are converted to character strings before output. The output is the line:

```
-7.09000000e+002xxx-3.54500000e+002
```

The first character of this line is in column 1 of the output medium.

The 'put' statement in this example requires further explanation. While the interpretation given for the statement is accurate, it may not be obvious how that interpretation was obtained. The two lists in an edit-directed statement are processed in parallel. That is, the sequence of processing is determined by the data list, but a reference is made to the format list for each item in the data list. When the next item in the format list is a data format item, such as 'a', it tells how the data value is to be processed, and the reference is complete. However, when the next item in the format list is a control format item, such as 'skip' or 'x(3)', it does not tell how to process the data item; and the control format item is executed and a new reference is made to the format list.

The program just given exercises a minimum of control over its input; it merely requires four characters that can be converted into an arithmetic value. Even though it was stated earlier that the input would be supplied as a signed, three-digit integer, this program accepts `'00+6'` or `'.003'` or even the very large number `'7e30'`. The program shows a similar indifference to the format of the output, and leaves this format entirely to the built-in rules for conversion of an arithmetic value to a character-string value.

The following version of the program uses pictured character-string variables to exercise the proper control of input and output:

```
P2: proc;
    dcl (sysin,sysprint) file;
    dcl in picture"s999";
    dcl out picture"-999.v9";
    get edit(in)(a(4));
    out = in/2;
    put edit(in,out)(skip,a,x(3),a);
end;
```

The variable name `'in'` is declared `'picture"s999"'`, and is thus restricted to precisely the signed, three-digit integer supplied as input when the program is properly used. Similarly, the variable name `'out'` is declared `'picture"s999v.99"'` to provide a good format for the computed result. If the string `'-7.9'` is supplied, the `'conversion'` condition occurs because this value cannot be assigned to `'in'`; and thus the error is detected. When the input is `'-709'`, the program runs to completion and outputs the line:

```
-709000-354.5
```

which is much more readable than the output from the first version.

The program just discussed controls input/output well, but uses pictured character-string values for its computation instead of the floating point variables used in the first version. This mode of computation causes a considerable loss of efficiency, especially if the computation is not so trivial. A third program combines well-controlled input/output with efficient computation:

```
P3: proc;
    dcl (sysin,sysprint) file;
    dcl in picture"s999";
    dcl out picture"-999.v9";
    dcl (x,y) float;
    get edit(in)(a(4));
    x = in;
    y = x/2;
    out = y;
    put edit(in,out)(skip,a,x(3),a);
end;
```

The version of the program just given uses the pictured variable names 'in' and 'out' in a rather specialized way; 'in' is an intermediary between the input stream and the computation-oriented variable 'x', and 'out' is an intermediary between 'y' and the output stream. PL/I has a special facility, the picture format item, for this situation, as the following, final, version of the program shows:

```
P4:  proc;
      dcl (sysin,sysprint) file;
      dcl (x,y) float;
      get edit(x)(p"s999");
      y = -x/2;
      put edit(x,y)(skip,p"s999",x(3),p"-999.v9");
      end;
```

This program means exactly the same thing as 'P3' did, and therefore needs no interpretation. Observe that it is identical to 'P1' except for the use of the picture format items instead of the 'a' format items.

The preceding examples have ranged from the simplest of the data format items, the character-string format item, to the most powerful, the picture format item. The examples have also shown two representatives, 'skip' and 'x', of the control format items. Finally, the examples have shown how the data list refers to the format list for the specification of format. The remainder of this discussion covers this ground again, supplying a complete description of the PL/I edit-directed input/output facility.

Data Format Items

In an edit-directed statement, each data item must have a corresponding data format item. Each data format item describes a field; that is, a sequence of characters either read from the input stream or added to the end of the output stream. Usually, the format item gives the width of the field; that is, the number of characters, including blanks, contained in the field. In addition, the format describes the way the value is represented in the field; that is, it gives the format of the value.

The format imposed on the input stream may or may not be very precise. For example, the format item 'p"s999' means, quite precisely, that "the next four characters of the input stream must be a signed integer with three digits". On the other hand, 'f(10)' means, less precisely, that "the next ten characters of the input stream must contain a fixed-point value representation, signed or unsigned, with or without a decimal point, filling the whole field or sharing it with leading or trailing blanks or both".

For output, the role of the format item is to supply the format of the output value representation. Again, the format may or may not be very precise. For example, 'e(20,8,9)' means "output four blanks and a floating point value representation that has a signed, nine-digit mantissa with eight digits to the right of the decimal point". On the other hand, the format item 'a' means "output however many characters are necessary to represent the given value".

The occurrence of linemarks and pagemarks in the stream are ignored during the processing of a data format item. During input, the field length may exceed the number of character positions that remain on the current line of the input stream; in this case, additional character positions in the subsequent lines are used. During output, the field length may require character positions beyond the number allowed by "linesize". In this case, new lines are created to supply additional character positions. If the new lines would exceed the number of lines allowed on the page by "pagesize", the "endpage" condition occurs and (when the condition has been processed) a new line is begun. Thus if the programmer wishes to ignore all or part of the layout of the output stream, he can do so and PL/I will process the layout automatically.

There is a format item for each of the main types of computational data. They are as follows:

<u>Name</u>	<u>Format Item</u>
character string	a(<u>w</u>)
bit string	b(<u>w</u>)
fixed point	f(<u>w</u> , <u>fw</u> , <u>dm</u>)
floating point	e(<u>w</u> , <u>fw</u> , <u>ms</u>)
complex	c(<u>part</u> , <u>part</u>)
picture	p" <u>x</u> "

In these format items, w (the width), fw (the fraction width), dm (the decimal multiplier), and ms (the mantissa significance) can each be any expression whose value can be converted to an integer. Except for the decimal multiplier, dm, the integer value must be positive or zero. Each part can be any format item for a real value. The x in the last format item represents a picture, as described earlier, in Section III, "Value Storage." Often it is not necessary to give all the arguments, as the following examples will show.

STRING FORMAT ITEMS

The stream representation of a string value is processed by the string format items. For character strings, the allowed forms are:

```
a(w)      -- used for both input and output
a          -- used for output only
```

For bit strings, the allowed forms are:

```
b(w)      -- used for both input and output
b          -- used for output only
```

The processing of the character-string and bit-string format items are parallel in every respect.

String Input

Examples of these format items for input follow:

<u>Input Stream</u>	<u>Format Item</u>	<u>Inbound Value</u>	<u>Comment</u>
2.5	a(4)	"2.5"	The next four characters are read,
	a(4)	"	whatever they are.
+3.0-2.3i	a(10)	"+3.0-2.3i"	The next ten characters are read.
010	b(3)	"010"b	Only '0' and '1' are accepted.
1	b(3)	"1"b	The blanks are deleted.
	b(3)	""b	The string can be empty.
000	b(3)	"000"b	Three zeroes are three bits.
012	b(3)	(conv)	Only bits, leading blanks, and
"01"b	b(5)	(conv)	trailing blanks can appear.

String Output

During output, a string value is left adjusted; that is, if the string value does not fill the field, it is placed as far to the left as possible and the remainder of the field is filled with blanks. This contrasts with the output of arithmetic values, which are right adjusted. When w is omitted from a string format item, the width of the output field is determined by the length of the output string value. This is the only case in which the field width is not given explicitly in a format item. Examples of the use of these format items for output follow:

<u>Outbound Value</u>	<u>Format Item</u>	<u>Output Stream</u>	<u>Comment</u>
"ABC"	a(5)	ABC	Blanks are supplied at the right.
"ABC"	a(5)	ABC	Given blanks remain in the string.
"	a(5)		A null string is accepted.
"ABC"	a	ABC	The value determines the width.
"1"b	b(5)	1	Blanks are supplied at the right.
""b	b(5)		No bits in a null bit string.
"00"b	b	00	The value determines the width.
"ABC"	a(5)	(stringsize)	Seven characters do not fit in the field.
"012"	b(3)	(conv)	Only bits are allowed for 'b(3)'. Only bits are allowed for 'b(3)'.

FIXED-POINT FORMAT ITEMS

The stream representation of a fixed-point value is processed by a fixed-point format item, which can have any of the following forms:

f(<u>w</u>)	-- used for input or output
f(<u>w</u> , <u>fw</u>)	-- used primarily for output
f(<u>w</u> , <u>fw</u> , <u>dm</u>)	-- used for special applications

The value of w (width) determines the size of the field. The value of fw (fraction width) determines the number of fractional digits in the representation. The value of dm (decimal multiplier) determines a multiplier for the transmitted value.

Fixed-Point Input

Usually the form 'f(w)' is used for input and the fraction-width and decimal-multiplier arguments are omitted. Input is performed as follows:

- A character string of w characters is read from the input stream. The string must contain an optionally-signed, real, fixed-point constant or else must be entirely blank.
- The input string is assigned to an intermediate variable of data type 'real decimal fixed(p,q)'. If (1) the input constant is 'decimal', (2) the input constant has a precision and scale-factor within the maximums of Multics PL/I, and (3) fw and dm are omitted from the format item, then the precision attribute (p,q) is taken directly from the input constant.
- The value of the intermediate variable is assigned to the target given in the data list.

In the steps just given, two assignments are made; one from the stream to an intermediate variable and a second from the intermediate variable to the target variable. These assignments are performed as if they arose from an assignment statement; that is, the necessary conversion is performed and, if the assignment or the conversion cannot be performed for some reason, the appropriate condition occurs. The intermediate variable exists only long enough to convey the value to the target, and is not used in any other way.

Examples of the processing of a five-character input field by a fixed-point format item follow.

<u>Input Stream</u>	<u>Format Item</u>	<u>Intermediate Value</u>	<u>Comment</u>	
-7.2	f(5)	-7.2	The position of the value representation in the field does not affect its interpretation.	
7.2	f(5)	+7.2		
07.2	f(5)	+7.2		
7.200	f(5)	+7.2		
00000	f(5)	+0.	If the field is blank, its value is 0.	
	f(0)	+0.		If <u>w</u> =0 there is no input, but value is 0.
72e-1	f(5)	(conv)	If the input is not a valid fixed-point value representation, the 'conversion' condition is signalled.	
-70.2	f(5)	(conv)		
7.2db	f(5)	(conv)		
72000	f(5,2)	+72	<u>fw</u> =2 gives two fractional digits, and <u>fw</u> =0 (default) gives none. But when a decimal point is in the stream, <u>fw</u> is ignored.	
72000	f(5)	+72.		
7.200	f(5,2)	+7.2		
7.200	f(5)	+7.2		
7.200	f(5,2,-3)	+0.0072	<u>dm</u> =-3 multiplies input by 10**(-3) = .001	
7.200	f(5,2,3)	+7200.		<u>dm</u> =3 multiplies input by 10**3 = 1000
72000	f(5,2,3)	+720.		<u>fw</u> =2 gives two fractional digits, then <u>dm</u> =3 multiplies by 1000.

The last two groups of examples show the use of a nonzero fraction width or decimal multiplier. Such format items should be used only when there is a clear justification for accepting input values that are not "true" values; this might occur, for example, when input is being prepared by an automatic device that can only produce a sequence of digits (but no decimal point or scale factor) on its output medium.

When fw is omitted, it is assumed to be zero; therefore, a value representation without a decimal point is treated as an integer, which is the everyday convention. When dm is omitted, it is also assumed to be zero; therefore, the value is multiplied by $10^{**0} = 1$. Thus the defaults are chosen so they leave the transmitted value unchanged.

Fixed-Point Output

Usually, the form 'f(w)' or 'f(w,fw)' is used for output, and the multiplier is omitted. When the form 'f(w)' is used, it is assumed that fw = 0. An intermediate variable is used with output in the same way as was previously described for input. The data type of the intermediate variable is 'real decimal fixed(p,q)'. The scale factor, q, is fw and the significance, p, is as large as w allows. That is, p is obtained by reducing w by one if necessary to allow for a minus sign and (when fw is not zero) by one more to allow for a decimal point; however, p cannot exceed the maximum Multics precision, 59. The outbound value is converted to an attractive, right-adjusted value representation in the output stream, as shown in the following examples:

<u>Outbound Value</u>	<u>Format Item</u>	<u>Output Stream</u>	<u>Comment</u>
-3.	f(5,2)	-3.00	<u>d</u> =2 provides two fractional digits, and
-3.	f(5)	ØØØ -3	<u>d</u> =0 (default) provides none.
-3.	f(5,3)	(size) -3.000	'-3.000' does not fit in the field.
+3.	f(5,3)	3.000	'3.000' (without sign) <u>does</u> fit.
+17.46	f(5,2)	17.46	<u>d</u> =2 fits the exact value (in this case),
+17.46	f(5,1)	Ø 17.5	<u>d</u> =1 rounds to one fractional digit, and
+17.46	f(5)	ØØØ 17	<u>d</u> =0 rounds to an integer.
+17.46	f(5,-1)	(error) (d	<u>d</u> must be nonnegative)
+0.	f(5,2)	Ø 0.00	There are many ways
+0.	f(5)	ØØØØ 0	to print zero.
+17.46	f(5,3,-1)	1.746	<u>dm</u> =-1 multiplies value by $10^{**-1} = .1$
+17.46	f(5,0,2)	Ø 1746	<u>dm</u> =2 multiplies value by $10^{**2} = 100$
+17.46	f(5,0,-1)	ØØØØ 2	then <u>fw</u> = 0 causes rounding.

The last group shows the use of a nonzero decimal multiplier with output. The use of this feature should be restricted to the applications that are similar to its use with input; that is, it should be used to change the "true" stored value of a number to some scaled output form for a specialized application.

FLOATING-POINT FORMAT ITEMS

The stream representation of a floating-point value is processed by the floating-point format item, which can have any of the following forms:

- e(w) -- used for input or output
- e(w,fw) -- used primarily for output
- e(w,fw,ms) -- used only for output

The value of w (the width) determines the size of the field. The value of fw (fraction width) determines the number of fractional digits in the mantissa of the representation. The value of ms (mantissa significance) determines the number of digits in the entire mantissa.

Floating-Point Input

Usually the form 'e(w)' is used for input and the fraction-width and mantissa-significance arguments are omitted. Input is performed as follows:

- A character string of w characters is read from the input stream. The string must contain (1) an optionally-signed, real constant that is either fixed-point or floating-point or (2) a sequence of blanks, which is interpreted as zero.
- The string is assigned to an intermediate variable of data type 'real decimal float(p)', where p is the precision of the input constant.
- The value of the intermediate variable is assigned to the corresponding target in the data list.

The role of the intermediate variable used here is the same as that used with the fixed-point format item. It exists only to convey the input to the target with the required conversions.

Examples of the processing of a seven-character input field by a floating-point format item follow:

<u>Input Stream</u>	<u>Format Item</u>	<u>Inbound Value</u>	<u>Comment</u>
1.3e7	e(7)	+1.3e+7	A floating-point or fixed point value representation is accepted at any position in the input field.
-50+4	e(7)	-50.e+4	
50	e(7)	+5.0e+0	
	e(7)	+0.0e+0	If the field is blank, its value is 0.
	e(0)	+0.0e+0	If <u>w</u> = 0, no input occurs, but value is 0.
1.3e7.0	e(7)	(conv)	If the input is not valid, the 'conversion' condition occurs.
-50+4	e(7)	(conv)	
13e7	e(7,2)	+13e7	<u>fw</u> = 2 gives two fractional digits, and <u>fw</u> = 0 (default) gives none. But when a decimal point is in the mantissa, <u>fw</u> is ignored
13e7	e(7)	+13.e7	
1.3e7	e(7,2)	+1.3e7	
1.3e7	e(7)	+1.3e7	

The last four examples show the use of a nonzero fraction width, fw. When an input constant does not contain a decimal point, fw supplies the position for an assumed decimal point. As in the case of the fixed-point format item, such floating-point format items should be used only when there is a clear justification for accepting input values that are not "true" values.

Floating-Point Output

For output, the data type of the intermediate variable is 'real decimal float(p)', where p is the precision derived from the data type of the value supplied by the data item. All three forms of the 'e' format are commonly used. Omitted arguments are interpreted as follows:

e(<u>w</u> , <u>fw</u>)	means	e(<u>w</u> , <u>fw</u> , <u>fw</u> +1)
e(<u>w</u>)	means	e(<u>w</u> , <u>p</u> -1, <u>p</u>)

Thus, when the mantissa significance is not specified in the format item, it is calculated so that the mantissa has a one-digit integer part. When neither fw nor ms is given, the precision of the output value itself, p, is used. The outbound value is converted to an attractive representation in the output stream, as shown in the following examples:

<u>Outbound Value</u>	<u>Format Item</u>	<u>Output Stream</u>	<u>Comment</u>
7.5	e(11,2,4)	Ø75.00e-001	When both <u>d</u> and <u>s</u> are given, the decimal can be adjusted.
7.5	e(11,0,4)	ØØ7500e-003	
7.5	e(11,3,3)	Ø0.750e+001	Note leading zero when <u>d=s</u> .
7.5	e(11,4,4)	0.7500e+001	Fits because there is no sign
7.5	e(11,3)	Ø7.500e+000	When <u>s</u> is not given, the mantissa has <u>d+1</u> digits, so there
-7.5	e(11,3)	-7.500e+000	is one integer digit
750	e(11,3)	Ø7.500e+002	Zero has a zero exponent.
0	e(11,3)	Ø0.000e+000	
+7.5e+0	e(11)	ØØØ7.5e+000	When <u>d</u> and <u>s</u> are not given, the precision is taken from the value itself.
+7.500e+0	e(11)	Ø7.500e+000	

COMPLEX FORMAT ITEMS

The stream representation of a complex value is processed by the complex format item, which can have either of the following forms:

c(part) -- used for input or output
 c(part1,part2) -- used for input or output

The part, part1, or part2 can be any format item for a real value; that is, one of the following:

fixed-point format item
 floating-point format item
 picture format item

This format item always describes two values in the stream: the real and imaginary parts of the complex value. If only one part is given, it is used twice. Although imaginary values are usually followed by 'i' in PL/I, this is not the case when the complex format item is used.

Complex Input

Examples of the use of this format item follow:

<u>Input Stream</u>	<u>Format Item</u>	<u>Inbound Value</u>	<u>Comment</u>
ØØ3Ø-2.3	c(f(3),f(5))	+3.-2.3i	The field is treated as two fields.
Ø3ØØØ-2.3Ø	c(f(5))	+3.-2.3i	
+2.32e-3-6.80e-3	c(e(8))	+2.32e-3-6.80e-3i	
Ø3ØØØØØØØØ	c(f(5))	+3.+0.i	A blank part is a 0.
Ø3ØØØ-2.3i	c(f(5))	(conv)	'i' is not allowed.

Complex Output

Examples of the use of the complex format item to process output follow:

<u>Outbound Value</u>	<u>Format item</u>	<u>Output Stream</u>
+3.0+2.3i +2.32e-3-6.80e-3i	c(f(5,2)) c(e(11,2),e(11,2,2))	Ø3.0Ø2.30 ØØ2.32e-002Ø-0.68e-002
+0.0e+0-3.0e+2i	c(f(8,2))	ØØØØ0.00Ø-300.00
+3.0-2.3i +3.0+2.3i	c(f(4,2)) c(f(4,2))	(size) 3.002.30

PICTURE FORMAT ITEMS

The stream representation of any computational value (except 'complex') can be processed by an appropriate picture format item, which has the form

p"x"

where x is any of the pictures described under "Pictured String Storage" in Section III, "Value Storage."

When input is performed under control of the picture format item, the following steps are performed:

- The length of the character string described by the picture is determined, and a string of that length is read from the input stream and assigned to an intermediate variable of data type 'picture"x"'.
• The value of the intermediate variable is assigned to the target variable.

Both of the assignments performed during the picture-format input require further comment. The first assignment changes the data type of the input string, but never changes the string itself. Suppose the input stream supplies the characters '-709' and the format item is 'p"s999"'; then the input value is '-709' and has the data type 'character'. After assignment to the intermediate variable, the value is still '-709', but it now has the data type 'picture"s999"'. Thus the string has been checked for conformity with the picture and has been given the interpretation associated with the picture.

The second assignment entails the conversion of the value of the intermediate pictured variable to the data type of the target value. Since the data type of the target can, in various cases, be any PL/I data type, there are many possible conversions.

When output is performed under control of a picture format item, the following steps are performed:

- The value supplied by the source expression in the data list is assigned to an intermediate variable of data type 'picture"x"', where x is the picture given in the picture format item.
• The value of the intermediate variable, which is already a character string, is added to the output stream without being changed.

Just as with input, two assignments are made; one from the source expression in the data list to the intermediate variable, and a second from the intermediate variable to the output stream. If an assignment cannot be performed, the appropriate PL/I condition occurs.

The pictures are classified into three groups: fixed-point, floating-point, and character. A fixed-point picture describes a character string that has the form of an optionally-signed, fixed-point constant. A floating-point picture describes a character string that has the form of an optionally-signed floating-point constant. A character picture describes a character string that might not be suitable for interpretation as an arithmetic value but could be useful as an identifier of some kind. Examples of picture format items from each classification are now given. The commentary given with the examples is an informal summary of the definitions given for pictures in Section III, "Value Storage."

Fixed-Point Pictures

The fixed-point picture format item is considered here, and many examples are given. For each '9' in a picture, a digit appears in the corresponding character position of the stream; for an 's', a sign ('+' or '-' but not a blank); and for a '-', a blank or a '-' (but not a '+').

<u>Stream</u>	<u>Format</u>	<u>Internal</u>	<u>Comment</u>
-823	p"s999"	-823.	A signed, three-digit integer with various signs.
+823	p"s999"	+823.	
Ø823	p"-999"	+823.	
-823	p"-999"	-823	

The examples above should be read in both directions. For example, the first line should be read first as the transmission of the characters '-823' from the input stream to a target, and then read a second time as the transmission of the value of an internal source with value '-823.' to the output stream. This approach will be used for subsequent examples of the picture format item.

A 'z' matches a digit; but if the digit would be a leading zero, it is suppressed (replaced by blank) on output and may or may not be suppressed (at the option of the user who prepares the data) on input. A sequence as 'sss' is like 'szz' except that the sign "drifts" to the right when leading zeroes are suppressed. The sequence '---' represents a "drifting" minus sign in the same way.

<u>Stream</u>	<u>Format</u>	<u>Internal</u>	<u>Comment</u>
ØØ68	zzz9	+0068.	Zero suppression and drifting signs for input or output
ØØØ5	zzz9	+0005.	
ØØØØ	zzzz	+0000.	
Ø+68	sss9	+0068.	
ØØ-5	---9	-0005.	

Observe that a blank field can appear in the stream, but only when there is no '9' in the picture.

In ordinary applications of the picture format item, the decimal point is indicated in the picture by two characters, 'v.', which matches the '.' in the stream.

<u>Stream</u>	<u>Format</u>	<u>Internal</u>	<u>Comment</u>
-53.60	sssv.99	-053.60	Ordinary decimal
53 -.60	sssv.99	-000.60	points for input
53 829.	----9v.	+00829.	or output.

When 'v' and '.' are not adjacent in the picture, the transmitted value is changed. The 'v' indicates the position of the decimal point in the internal representation of the value, and the '.' indicates the position of the decimal point in the stream representation. Details are given in Section III, "Value Storage."

A parenthesized integer can be used in a picture to indicate repetition of the following picture character. For example,

```
p"s(7)9"      means  p"s9999999"
p"(5)sv.(2)9" means  p"sssssv.99"
```

The parenthesized integer must be a constant; that is, it cannot be written as an expression and computed when the program is executed.

When one of the assignments in the interpretation of a picture format item would lose a digit at the left end of the value, the 'size' condition occurs. But when an assignment would lose a digit at the right end of the value, that digit is truncated, without warning, and no condition occurs. Suppose the target of input or the source of output has the data type 'dec(6,2)'. The following examples show various instances of digit-loss:

<u>Stream</u>	<u>Format</u>	<u>dec(6,2)</u>	<u>Comment</u>
-9.2362	p"-9.9999"	(size)	Input error.
-9.2362	p"-9v.999"	-0009.23	Input approximation.
(size)	p"s999"	+8264.00	Output error.
+82	p"s99"	+0082.99	Output approximation.

The most remarkable aspect of the picture is its handling of commercial symbols. A '\$' can be used and can "drift" to the right as a sign can do. Commas can be used and a comma is suppressed when an adjacent leading zero is suppressed. The suffixes 'cr' and 'db' are allowed.

<u>Stream</u>	<u>Format</u>	<u>Internal</u>	<u>Comment</u>
\$129.88	\$999v.99	+129.88	Commercial symbols for input and output.
00 \$6.50	\$\$\$9v.99	+006.50	
\$2,619	\$\$,\$\$\$	+2619.	
000 \$81	\$\$,\$\$9	+0081.	
\$9.28cr	\$9v.99cr	-9.28	

These examples of the fixed-point picture format items do not exhaust all the possibilities, but the omitted possibilities are less frequently used. For a complete description of fixed-point pictures, see Section III, "Value Storage."

Floating-Point Pictures

The mantissa of the floating-point picture can be any fixed-point picture that does not contain the commercial symbols, '\$', 'cr', and 'db'. The exponent picture can have 's' or '-' as its sign or the sign can be omitted; and up to three digits can be used, with 'z' for leading digits if desired. If an 'e' is not wanted between the mantissa and the exponent, 'k' is used in the picture instead of 'e' and nothing appears in the stream.

<u>Stream</u>	<u>Format</u>	<u>Value</u>
+3.939e+002	p"s9v.999es999"	+393.9
00 3.939e+02	p"-9v.999es99"	+393.9
00 3.939+02	p"-9v.999ks99"	+393.9
00 3939-02	p"-9999ks99"	+39.39

Character Pictures

A character picture can be any sequence of the characters 'x' (matches any character), 'a' (matches any letter, upper or lower case, or blank), or '9' (matches any digit or blank). The picture must not be all nines, since it would be a fixed-point picture in that case.

<u>Stream</u>	<u>Format</u>	<u>Value</u>	<u>Comment</u>
3/may/74	p"9xaaax99"	"3/may/74"	Character string, input and output
29AXQ6	p"99aaax"	"29AXQ6"	
000000	p"99aaax"	" 000000 "	

A '9' can match a blank only in a character picture.

Control Format Items

In an edit-directed statement, provision must be made for those positions of the stream that appear between the value representations. Specifically, the contribution to layout made by the blanks, linemarks, and pagemarks must be taken into account. The control format items are provided for this purpose. Two examples of control format items were given in the example program, namely 'skip' (start a new line) and 'x(3)' (skip three character positions).

An output stream with the attribute 'print' can be viewed as divided into pages and lines and character positions. Any other stream, whether for input or output, is divided only into lines and character positions within the lines. A character string that is used as a substitute for a stream (by means of the 'string' option of a stream input/output statement) is a single line that is divided only into character positions.

When an input stream is open, it has a stream pointer associated with it. The stream pointer indicates the next character position that will be read (or skipped) by the next input operation. In contrast, an output stream is created as output is performed; that is, character positions as well as the characters themselves are added to the end of the stream. But it is legitimate and very useful to speak as if the output were created in advance as a sequence of blank character positions arranged in lines and pages. This convention allows the use of a stream pointer with an output stream and permits language such as "advance the stream pointer to the first character position in the third line after the current line".

There are five control format items, as follows:

x(<u>e</u>)	--	skip <u>e</u> character positions
column(<u>e</u>)	--	skip to column <u>e</u> of a line
skip(<u>e</u>)	--	skip <u>e</u> lines
line(<u>e</u>)	--	skip to line <u>e</u> of a page
page	--	skip to the next page

The e in each of these format items can be any expression whose value can be converted to an integer. In all cases, the value of e must be positive or (for 'x' and 'skip' only) zero.

'x' FORMAT ITEM

The 'x' format item has the form

x(e)

Let n be the value of the expression e for a given execution; then the item moves the stream pointer forward by n character positions, proceeding from line to line or page to page if necessary. If n=0, then the item does nothing.

'column' FORMAT ITEM

The 'column' format item has the forms:

```
column(e)  
col(e)
```

Let n be the value of the expression e for a given execution; then the item advances the stream pointer to the next character position that is in column n; that is, to a character position that is the nth character position of a line. This interpretation implies that if the stream pointer is beyond the nth column the operation is applied to the next line. If n exceeds the length of the line (so the specified character position does not exist), the stream pointer is set to the beginning of the next line.

'skip' FORMAT ITEM

The 'skip' format item has the form:

```
skip(e)
```

Let n be the value of the expression e for a given execution; then the item moves the stream pointer to the first character position of the nth line after the current line. If n = 0, then the stream pointer is set back to the beginning of the current line and the stream is prepared for overprinting of the current line; but this case is allowed only for an 'output print' stream.

'line' FORMAT ITEM

The 'line' format item has the form:

```
line(e)
```

Let n be the value of the expression e for a given execution; then the item moves the stream pointer to the next character position that is the first character position of the nth line of a page. If the stream pointer is already at such a character position, the stream pointer is not moved. If the stream pointer is already past the target position, or if the target position is beyond the end of the page, the stream pointer is advanced to the beginning of the next page.

'page' FORMAT ITEM

The 'page' format item has the form:

```
page
```

The item moves the stream pointer forward to the first character position of the next page.

A control format item can be used only where its use would be reasonable. Any control format item can be applied to an output stream with the 'print' attribute because it has lines and pages. The 'line' and 'page' items cannot be applied to a stream that is not a 'print' output stream because such a stream is not divided into pages. The 'skip', 'line', and 'page' items cannot be applied when the 'string' option is used because a pseudo-stream is not divided into pages or lines.

A control format item is executed only when PL/I is "on the way" to a data format item; that is, when PL/I is prepared to output a value and is reading through the format list toward the next format data item.

Format Lists

In its simplest form, a format list is a sequence of format items separated by commas. However, there are three facilities for enhancing the form of a format list; namely, the remote format item, the iterated format item, and the "end-around" repetition. These facilities are discussed in the following paragraphs.

REMOTE FORMAT ITEMS

The remote format item has the form:

r(ref) -- use a remote format list

The ref must be a reference that has a scalar format value; that is, a value that designates a 'format' statement. Let 'fx' be the format list in the designated 'format' statement. When 'r(ref)' is executed, the scanning of the format list in which the remote format item appears is suspended and format items are taken from 'fx' until the end of 'fx' is reached.

ITERATED FORMAT LISTS

The iterated format list can have any of the following forms:

<u>int</u>	<u>item</u>	-- constant iteration of an item
<u>int</u>	(<u>fl</u>)	-- constant iteration of a format list
(<u>e</u>)	<u>item</u>	-- computed iteration of an item
(<u>e</u>)	(<u>fl</u>)	-- computed iteration of a format list

In these forms, int is an unsigned integer, item is a data, control, or remote format item, fl is a format list, and e is any expression whose value can be converted to an integer. Suppose the value of int or e (whichever is present) is n. Then the iterated format list is interpreted as a sequence of format items composed of n repetitions of item or fl (whichever is present). If n is zero, the iterated format list is ignored.

END-AROUND REPETITIONS

The "end-around" repetition is a simple feature of the edit-directed statements. An outermost format list in an edit-directed statement is repeated when the end of the list has been reached. An outermost list is the format list paired with a data list in the statement. The effect of this convention is that the format list can never "run out" before the corresponding data list does.

The program fragment that follows shows the use of the compound format items:

```
...
F5: format(a(10), 2(p"bbb--9v.99"));
...
i = 5;
put(...)(page, (i-2)(skip, r(F5), col(20), a));
...
```

The format list in the 'put' statement is equivalent to the following format list:

```
page,
skip, a(10), p"bbb--9v.99", p"bbb--9v.99", col(20), a,
skip, a(10), p"bbb--9v.99", p"bbb--9v.99", col(20), a,
skip, a(10), p"bbb--9v.99", p"bbb--9v.99", col(20), a,
page,
skip, a(10), p"bbb--9v.99", p"bbb--9v.99", col(20), a,
... and so on, ad infinitum.
```

In this format list, the data format items have been underlined to distinguish them from control format items. It is the data format item that is matched with each item in the data list, so the portion of the format list shown would accommodate 16 items from the data list.

The edit-directed statement uses the same form of data list as the list-directed statement; and the interpretation of that list to produce a simple list of data items was given in the discussion of list-directed input/output. Now, immediately above, the interpretation of a format list to produce a simple list of format items has been given. On the basis of these interpretations, the items of any data list can be matched to the items of the corresponding format list.

Guidelines for Edit-Directed Input/Output

Edit-directed input/output is preferred whenever the programmer wants to assume control over the format of input or output. It provides a wide variety of facilities for specifying format; and even within edit-directed techniques there is a range of control over details. At one extreme, the programmer can use the fixed-point format item and require, in a rather indefinite way, that an optionally-signed constant appear in certain columns of a line. At the other extreme the programmer can use a picture and control the contents of a line on a character-by-character basis.

The format list associated with an edit-directed statement can easily become complicated and unintelligible. It is important that a layout diagram be made of the document being read or written, and that the format-list be based on this diagram. The 'format' statement can be used to structure a complicated format list just as the procedure is used to structure a complicated program.

The three disciplines of stream input/output can be mixed. For example, certain codes at the beginning of an input stream could be read by an edit-directed 'get' statement, and then a specified number of values could be read by a simple list-directed statement. However, care must be taken when switching back to edit-directed input. Since edit-directed input works on a strict column by column basis, a serious error can occur if the effect of the preceding non-edit-directed statement is not correctly determined.

STRING OPTION

In order to interpret the input/output statements, PL/I must have a large and complicated collection of string manipulation operations. In particular, the process of applying a format list to the input stream to produce values or of applying a format list to a value list to produce an output stream is a complicated operation. Accordingly, PL/I has a facility to make this string manipulation available independent of the performance of input/output.

A 'get' statement can have an option of the form 'string(e)' instead of the usual file option, where e is any character-string expression. In this case, the statement will take its input from the value of e as if that value were a complete stream data set. Similarly, a 'put' statement can have an option of the form 'string(t)' instead of the file option, where t is any target that can accept a character-string value. In this case, the statement will assign its entire output to t as if that target were a stream data set.

Linemarks and pagemarks cannot be used when the 'string' option is used. If a 'get' statement with a 'string' option "runs off the end" of the pseudo stream, the 'error' condition occurs rather than the 'endfile' condition. Thus the extension of input/output statements to the use of the 'string' option applies only to the editing process itself and not to those aspects that are oriented toward input/output.

A useful application of the string option arises in connection with a troublesome property of stream input: the input of characters cannot be controlled by anything that appears later in the stream. Consider an example of this problem. Suppose 80-character card-images are being read and they can occur in either of two formats depending on whether an '*' or a blank appears in column 80. This problem can be solved by using the following statements:

```

.....
get edit(temp)(a(80));
if substr(temp,80,1) = "*"
    then get string(temp) edit(C1, C2, C3)(p"$$$$$v.99db", X(7));
    else get string(temp) edit(C1, C2, C3)(p"$$$$$v.99-", X(8));
...

```

If a card ends with '*', this sequence of statements is equivalent to:

```
get edit(C1, C2, C3)(p"$$$$$v.99db");
```

and otherwise the sequence is equivalent to

```
get edit(C1, C2, C3)(p"$$$$$v.99-");
```

The use of the string option allows the program to "look ahead" in the input stream and select a format appropriate to the coming values.

SPECIAL FEATURES

In Multics PL/I the sequential 'read' and 'write' statements can be used with stream data sets to provide special "line at a time" processing. In this special processing mode, each line (a group of characters terminated by a linemark) is treated as a record on input and each record written has a linemark appended.

One very important point is that the PL/I rules for implicit opening have not changed. Therefore, a user wishing to use the 'read' or 'write' statements must explicitly open the stream using the 'open' statement as described in "Opening and Closing Files" earlier in this section.

'read' Statement

Special stream input is performed by the 'read' statement when the file that is read is a 'stream input' file. For example:

```
read file(test2) into(my_string);
```

The file 'test2' must be opened for 'stream input' by the 'open' statement. The 'read' statement must include an 'into' option specifying a scalar character-string variable and cannot contain a 'key' option.

The example statement assigns all characters up to, but not including, the next linemark or the end of file to 'my_string'. If no characters remain between the 'stream pointer' and the end of file, the 'endfile(test2)' condition is signalled. The 'stream pointer' is then placed past the linemark, if any. If the assignment of the characters in the input stream to 'my_string' would cause the 'stringsize' condition to be signalled, the 'record(test2)' condition is signalled instead. It is recommended that the 'into' option specify a varying character string for ease of use.

'write' Statement

Special stream output is performed by the 'write' statement when the file written to is a 'stream output' file. For example:

```
write file(report) from(source_string);
```

The file 'report' must be opened for 'stream output' by the 'open' statement. The 'write' statement must include a 'from' option specifying a scalar character-string variable and cannot contain a 'keyfrom' option.

This 'write' statement appends the characters of 'some_string' and a single linemark to the output stream 'report'.

If the length of 'some_string' is greater than the 'column position' plus 'line size' the 'record(report)' condition is signalled. Upon return from the 'on' unit, the number of characters written are the first 'line size' minus 'column position' plus one character. Finally, the 'line number' is incremented by one and if it is equal to the 'page size' plus one, the 'endpage(report)' condition is signalled.

CONDITIONS FOR STREAM INPUT/OUTPUT

In the following discussion, the conditions that occur during stream input/output are described. They are:

```
conversion
endfile(ref)
endpage(ref)
name(ref)
transmit(ref)
undefinedfile(ref)
```

where ref is a reference that yields a file value. The general rules for the use of the conditions are given earlier, in Section XIII, "Condition Handling." Only some remarks about their application to input/output will be given here.

Each condition is defined separately for each file value, and thus for each file state block. The identifier 'endpage' by itself is not a valid condition; but if 'record3' is a file constant name, then 'endpage(record3)' is a valid condition. Consider the statement

```
on endpage(record3) put file(record3) page line(3);
```

When this statement is executed, it establishes the 'put' statement as the 'on' unit for the condition 'endpage(record3)'. When the end of a page in the output stream associated with 'record3' is reached, the 'endpage' is signalled and the 'on'-unit is executed. When the block that contains the 'on' statement is deactivated, the 'on' unit is reverted, and no longer responds to a signal.

When a condition is signalled, the PL/I processor takes either of two actions, as follows:

- If an 'on' unit is established for the condition, then that 'on' unit is executed. If the execution of the 'on' unit runs to completion, then control goes back to the point in the program at which the condition occurred, and execution is resumed in a reasonable way (depending on the particular needs of the statement involved).
- If no 'on' unit is established for the condition, then the default 'on' unit is executed. The default 'on' unit for each condition is described earlier, in Section XIII, "Condition Handling."

A stream input/output statement can evaluate expressions, and during that process a 'fixedoverflow', 'overflow', 'underflow', or 'zerodivide' condition may occur. Further, a stream input statement assigns values to targets, and during that process a 'size', 'stringrange', 'stringsize', or 'subscriptrange' condition may occur.

This page intentionally left blank.

The PL/I processor saves certain useful values before signalling a condition. For each kind of value saved, there is a stack and a built-in function. Just before the condition is signalled, the value is placed on the top of the stack, and after completion of the established 'on' unit it is removed. The built-in function is used to access the value during the execution of the 'on' unit.

For example, just before any of the conditions mentioned in this section is signalled, the file name, expressed as a character-string value, is placed at the top of the stack controlled by the 'onfile()' built-in function. During the execution of the established 'on' unit for the condition, the file-name character-string can be accessed by using the reference 'onfile()'. When execution of the 'on' unit is complete, the file name character string is removed from the stack.

'conversion' Condition

The 'conversion' condition occurs when an attempt is made to convert an invalid character string or pictured value to an arithmetic or bit-string value. Just before the condition is signalled, three values are saved in the stacks controlled by the condition built-in functions. The character string being converted is placed at the top of the stack controlled by 'onsource()'. The leftmost character in the string at which conversion failed, which is sometimes the source of the error, is placed at the top of the stack by 'onchar()'. The file name is saved as described in the preceding paragraph.

The 'onsource()' and 'onchar()' functions can be used as pseudo variables, and the 'on' unit can assign new variables to them; in this way, it is possible to "correct" a character string that is causing trouble. When a normal return from the 'on' unit occurs, the PL/I processor resumes its attempt to convert the offending character string. If the program has supplied a new and valid value by means of 'onsource()' or 'onchar()' then the conversion succeeds, and execution continues; otherwise, the 'conversion' error occurs again.

'endfile' Condition

The 'endfile' condition occurs when an input statement attempts to read beyond the end of a data set. After an established 'on' unit is executed, the PL/I processor resumes with the statement after the input statement in which the condition occurred. If a later attempt is made to read the data set, the condition will occur again. The file name is saved in the stack controlled by 'onfile()'.

'endpage' Condition

The 'endpage' condition occurs when an output statement completes the n th line of a page (by writing a linemark) and n is equal to the "page size" associated with the output file. The condition can be caused in either of two ways, and the action taken by the PL/I processor on return from an established 'on' unit varies accordingly. If the condition was caused by an attempt to write a data value in the output stream, the output of the data is completed when execution resumes. But if the condition was caused by the interpretation of a 'skip' option or format item or a 'line' option or format item, then the option or format item is ignored; it is assumed that the 'on' unit starts a new page and eliminates the need for the blank lines.

When the 'endpage' condition is signalled, the line number associated with the file has already been increased by one and is therefore equal to the 'page size' plus one. Normally, the 'on' unit will include a 'page' option or format item and will thereby set the line number back to 1.

Just before the 'endpage' condition is signalled, the file name is saved in the stack controlled by the 'onfile()'. If there is no established 'on' unit for the condition, the PL/I processor does not treat the condition as an error; instead, a pagemark is added to the output stream, the line number is set to 1, and execution of the program continues.

'name' Condition

The 'name' condition occurs only during data-directed input. Specifically, the condition occurs when a stream assignment is read whose variable name does not match a variable name in the data list of the controlling 'get' statement or a name of a component of a variable that is named in the data list. If the 'string' option is not specified, then before the condition is signalled, the offending assignment from the stream is placed at the top of the stack controlled by 'onfield()', and the variable is therefore available as a character-string value for inspection with the 'on' unit. The file name is placed at the top of the stack controlled by 'onfile()'. After an established 'on' unit is executed, the PL/I processor returns to the data-directed input as if the processing of the offending stream assignment were complete.

'transmit' Condition

The 'transmit' condition occurs when data cannot be transmitted reliably between a data set and PL/I storage. Just before the condition is signalled, the file name is placed at the top of the stack controlled by 'onfile()'. After an established 'on' unit is executed, the PL/I processor resumes with the statement that follows the input/output statement that caused the condition; but the value of the data transmitted by the statement is undefined.

The condition is usually caused by factors beyond the programmer's control, such as a hardware failure, so the recovery procedure cannot be initiated until the hardware is repaired.

'undefinedfile' Condition

The 'undefinedfile' condition occurs when an 'open' statement attempts unsuccessfully to open a file. The condition can occur, when, for example, an attempt is made to open a record data set for stream input, or when, for another example, the 'title' option specifies an invalid attachment or a nonexistent file. Just before the condition is signalled, the file name is placed at the top of the stack controlled by 'onfile()'. After the established 'on' unit is executed, the program resumes execution at the statement following the offending 'open' statement.

SECTION XV

RECORD INPUT/OUTPUT

The record input/output facility of PL/I is independent of the stream input/output facility described in the preceding section; that is, it has its own data sets, statements, and programming techniques. The record input/output facility is oriented toward communication with permanent storage. The role of such storage is to accept values from PL/I at one time and then return them, unchanged, at a later time; therefore, each value is transmitted just as it is found in PL/I storage. In contrast, stream input/output is oriented toward user communication and has many ways of converting between internal values and external representations of those values.

The orientation of record input/output toward communication with permanent storage does not prevent its being used for communication with the user. Since PL/I has a capacity for conversion of values and string manipulation that is independent of any input/output operations, it is possible and even convenient to prepare user-oriented character strings before an output operation is initiated. Once such a character string has been prepared, it can be transmitted as a record to a printer or a terminal. The same considerations apply to input. Thus record input/output can handle user communication as well as permanent computer storage.

This section begins with a description of the two kinds of data that are involved in record input/output: the record data set, which is the actual subject of the input or output, and the file-state block, which shows the status of the operations on the record data set. The section then describes the attachment of a PL/I data set to a Multics file. In order to make this section complete and independent, the description of the file-state block and file attachment repeats some material already given in Section XIV, "Stream Input/Output." The section continues by giving a summary of the operations that are performed as a part of record input/output. Once this foundation has been established, the section proceeds to a definition of the statements that are used for record input/output: first, the statements that open and close files and then the statements that perform the actual input/output operations. Next, the section describes based input/output, which is an advanced and specialized feature of record input/output. As the section nears completion, the use of record input/output for user communication is illustrated. Finally, the section describes the conditions that occur in connection with record input/output.

RECORD DATA SETS

A record data set is a collection of records. Each record is a single PL/I value; that is, it is a copy of a value that once existed in PL/I storage. The record can be a single scalar value; indeed, it can be a "bit(1)" value and thus represent only one bit. On the other hand, the record can be an aggregate value such as a large and complicated structure or an array of many elements. Some of the costs of transmitting and storing a record are the same for records of all sizes; therefore, large records are preferred. For example, if a programmer has a choice between treating an array as a single record or treating each element of the array as a record, then he should choose the first alternative.

The word "record" is used here to mean a logical record; that is, a collection of information gathered together because it belongs together. Hardware storage devices do have physical records; that is, units that reflect the architecture of the storage device. The relation of the PL/I logical record to the physical record is similar to the relation of the PL/I variable to the hardware computer word. In both cases, PL/I provides an elaborate and effective mechanism to allow a programmer to choose units that correspond to the logical requirements of the data and to ignore the boundaries that are built into the hardware.

Organization Of Record Data Sets

A record data set can be keyed, sequential, or keyed sequential. In a keyed data set, each record has a unique key associated with it that can be used to access the record directly without scanning through the file. In Multics, the key is a character-string value of length up to 256 characters. In a sequential data set, the records are arranged in an order that does not change and that can be used to pass from one record to the next when the file is being processed. In a keyed sequential data set, a record can be accessed either by its key or by its sequential position. The organization of a data set determines the kinds of operations that can be performed on it.

When a file is being operated on, it has two indicators associated with it. The current record indicator designates the record that has been most recently operated on. The next record indicator designates the record that will be read if the next operation is a sequential read operation; it is defined only for a sequential file. Unless otherwise stated, whenever the current record indicator is reset, the next record indicator is adjusted to designate the next record in sequence. Under certain circumstances, an indicator is set to null (and does not point to any record); for example, when current record indicator is set to designate the last record of a file, the next record indicator becomes null.

Multics Files

There are a variety of ways to implement a record data set, each reflecting different hardware requirements and software techniques. Multics has two implementations for a record data set, the sequential and the indexed files.

A sequential Multics file can be used for an unkeyed sequential PL/I data set. Its records are arranged in the order in which they are created, and a record can be rewritten only if the new value has the same storage type as the old value. A sequential file is either in virtual memory or on a magnetic tape.

An indexed Multics file can be used for any keyed PL/I data set, sequential or not. Its records are arranged in order of ascending keys. That is, if the key k₁ precedes the key k₂ in a file, then the relation $k_1 < k_2$ (as defined for PL/I character strings) must be true. A record can be rewritten in any way; that is, the storage type of the new record need not conform to that of the old record. An indexed Multics file is always stored in virtual memory.

RECORD FILES

A connection must be established between a statement that performs input/output and the Multics file on which the operation is to be performed. An analysis of this connection follows:

- The connection begins with the file option that appears in an input/output statement.
- The file option has as its argument a file reference, and the evaluation of the file reference yields a file value.
- The file value designates a file-state block, which is a set of values that are used by the PL/I processor in carrying out input/output operations.
- The file-state block contains a data set designator that points to a Multics file and thus completes the connection between input/output statement and file.

The main components in the connection just described are the file-state block and the file reference; these components are described in the following paragraphs.

First, however, a problem of terminology must be resolved. In PL/I, the source of input and the destination of output is called a data set; but in Multics, it is called a file. This difference is observed when it is necessary to distinguish between the PL/I view of input/output, as in "a keyed sequential data set", and the Multics view, as in "an indexed Multics file". The word "file" is also used as a PL/I term, and in that usage, it refers to the combination of the file-state block and the data set; thus, the phrase "open a file" actually refers to the setting of a certain file-state block to control input/output with a certain data set.

File-State Blocks

Transmission of values between the PL/I processor and a Multics file requires bookkeeping data. This data is stored in a portion of system storage called a file-state block. When a file is open, the file-state block contains the designator of a Multics file and other information about input/output in progress. After the file is closed, the only information in the file-state block that is meaningful is that supplied by the attributes, if any, in the declaration of the file constant name. A file-state block cannot be accessed directly, but its values are changed when input/output is performed on the data set with which it is associated.

The following values in a file-state block are relevant to record input/output:

- The status indicator. This value shows whether the file-state block is open or closed.
- The data set designator. This value points to the Multics file that is associated with the file-state block.
- The file name. This value is a character string that is the identifier that is the name of the constant file value that designates the file-state block.
- The file attributes. These attributes are those associated with the current use of the file-state block.
- The current record and next record indicators. These values point to the current position of input/output operations within the given data set.

In addition to the items just listed, there are other items, such as buffers, that are not of immediate interest to a programmer.

File References

A file-state block is designated by a file value, and the file value is supplied by a file reference in a 'file' option. The file reference can be a reference to a constant, a variable, or a function.

A file constant reference is a name that has been declared with the following attributes:

[external]
[internal] file [constant]

The default rules provide that the scope attribute can be omitted if it is 'external'. The 'constant' attribute can be omitted in any case, as indicated by the square brackets.

Every file constant name must have an associated file description. It is recommended that this file description be given when the file is opened, as described later in this section, under "The 'open' Statement". However, PL/I does allow the programmer to write any portion of the file description attributes in the declaration of the file constant name.

Each declaration of a file constant name associates the name with its own file-state block in static system storage. The only exception is the declaration of a given name in several different blocks as 'external file constant'; in this case, the declarations all refer to a single file-state block, as is required by the interpretation of the 'external' attribute. A given file constant name and its associated file-state block can be used for more than one data set in the course of a process by any number of PL/I programs. For example, a file-state block can be opened for input from a stream data set, closed, opened for updating a record data set, closed again, and so on.

A file variable reference or a file function reference is similar to a variable reference or a function reference of any other type. However, two exceptional features of file variable names are:

- The default scope of a 'file' variable name is 'external', whereas the default scope for most other variable names is 'internal'.
- The attribute 'variable' must be used explicitly for a file variable name because the default for a name of type 'file' is 'constant'.

FILE ATTACHMENT

The Multics I/O system uses a software construct, the I/O switch, to control the source or destination of an input/output operation. A PL/I file-state block is attached to a Multics file through a named switch. The switch name and the file name are the same for an external file. For an internal file, a unique name is generated for the switch.

Two operations, attachment and opening, are associated with I/O switches. When an I/O switch is attached, the source or target and the I/O module that performs the input/output are established. When an I/O switch is opened, a particular mode of processing is established.

Attaching a Switch

An I/O switch can be attached either at command level by the 'io_call' command or within a PL/I program by the execution of an input/output statement. If the switch is not attached when the 'open' statement for the associated file is executed, the information in the 'title' option is used to attach the switch. An I/O switch attached by the execution of an 'open' statement for a file is detached when the 'close' statement for the file is executed. If an I/O switch is already attached, neither the 'open' nor the corresponding 'close' statement has any effect on the switch's attachment.

ATTACH DESCRIPTION

The 'title' option contains the attach description. The attach description specifies an I/O module to perform the input or output operation and the file or device to be used as the source or destination for these operations. For record input/output the following I/O modules can be used:

<u>I/O Module</u>	<u>Usage</u>
vfile_	for storage-resident files
syn_	for synonym attachment
record_stream_	for conversion between record and stream files
tape_ansi_	for tape files
tape_ibm_	for tape files

The form of the attach description depends upon the I/O module. For complete details on the attach description of a particular I/O module, see the MPM Subroutines or MPM Peripheral I/O. The principal features of the I/O modules, `vfile_`, `syn_`, and `record_stream_` are described later, in Section XVI, "PL/I in the Multics System." As an example of an attach description, consider the following 'open' statement:

```
open file(recl) title("vfile_ beta") sequential output;
```

The attach description in the 'title' option specifies that the I/O module `vfile_` is to be used to perform output to the system-resident file in the segment 'beta'.

If no 'title' attribute is given, a default attach description is formed, as follows:

```
vfile_ fn
```

where fn is the file name.

Opening a Switch

An I/O switch can be opened either at command level by the `io_call` command or within a PL/I program by the execution of an input/output statement for the file associated with the switch. If the switch is not open when an 'open' statement is executed for the file, the information in the file description is used to open the switch.

The file description attributes specify the opening mode of the switch. A switch's opening mode must be compatible with its attachment. All opening modes are compatible with the `vfile_` I/O module. For information concerning opening modes compatible with other I/O modules, see Section V of the MPM Reference Guide for a table giving a list of all opening modes supported by I/O modules. For individual descriptions of I/O modules, see MPM Subroutines or MPM Peripheral I/O or MPM Communications I/O.

Opening a File

A PL/I file-state block, or file, is opened by the execution of the first input/output statement referencing the file. The file name, title, and file description are passed to the Multics I/O System to open the file. If any of these options is not explicitly given, a default assumption is derived from the input/output statement.

The Multics I/O system uses the title to attach the switch if it is not already attached and the file description to open the switch if it is not already open. Then the Multics I/O system returns a data set designator. This data set designator makes the connection between the PL/I data set and the Multics file. The data set designator is stored in the file-state block for use when an input/output operation is performed on that file.

RECORD INPUT/OUTPUT OPERATIONS

A summary of record input/output operations is given here. It introduces terminology, shows how data sets are manipulated, and gives a general view of the record input/output facility.

When a data set is opened for output, the contents of the data set are discarded and the data set is ready to accommodate the writing of new records. When a data set is opened for input, the contents are retained and the data set is made available for reading of its records. When a data set is opened for update, the contents are retained and the data set is made available for reading, writing, deleting, or rewriting of records. When a sequential data set is opened for 'input' or 'update', the next record indicator is set to designate the first record of the data set. The contents of a data set remain accessible to PL/I in this way until the data set is closed.

A given input/output operation uses either the keyed or the sequential properties of a data set, but not both; and this distinction is useful in the description of record input/output. A keyed operation uses the key supplied by an input/output statement to find the record to be operated on. A sequential operation uses the current record or next record indicators for this purpose.

When a record is created and assigned a value it is said to have been written. A keyed write operation places the new record and its key in its proper sequential position to maintain the ascending sequence of keys. An unkeyed write operation places the new record at the end of the data set. In either case, the value is copied into the record exactly as it appears in the referenced variable in PL/I storage.

A keyed read operation begins by locating the record that has the specified key and designating it as the current record. A sequential read operation begins by designating the next record as the current record (and thus advancing by one record). In either case, the value of the current record is then copied into the designated unit of PL/I storage. If the storage type of the record and the storage unit are not identical, the operation is invalid.

A keyed delete operation begins by locating the record that has the specified key and designating it as the current record. A sequential delete operation begins by finding the current record. In either case, the current record is then discarded, with the result that there no longer is a current record; that is, the current record indicator is set to null.

A rewrite operation replaces an existing record with a new record. If the data set is an unkeyed 'sequential' data set, then the new record must have exactly the same storage type as the old record. For a keyed 'indexed sequential' data set, there is no such restriction.

The based input/output operations are a relatively specialized facility. When based input is performed, PL/I automatically allocates storage with storage type identical to the record; and thus a record can be input even when its storage type cannot be predicted by the programmer.

A broad spectrum of errors can occur during record input/output. An attempt to modify a record in a data set that was opened for output is a programming error. The input of a record whose structural attributes do not agree with the designated PL/I storage unit may be an input-data error. Inaccurate transmission of a value between a data set and PL/I storage is a system error. And finally, an attempt to read beyond the end of a sequential data set may not be an error at all but rather a convenient way of ending an input loop. PL/I detects these conditions when they occur and the programmer can provide an "on" unit to respond to each condition with suitable actions.

OPENING AND CLOSING FILES

When a file is opened, the file-state block is marked "open" and the data set designator, control parameters, and indexes are set in the block. When a file is closed, the file is marked "closed" and only information provided by the file declaration is meaningful.

A file is opened when the first input/output statement referencing the file is executed. The purpose of the 'open' statement is to provide the title and file description for the file opening. However, both these options can be omitted from the 'open' statement, and, in that case, a default assumption is made. If an 'open' statement is not given for a file, the attributes for the file opening are derived from the first input/output statement executed. If a file is already open when an 'open' statement is executed, the 'open' statement is completely ignored.

'open' Statement

An 'open' statement gives a file value, a title for a Multics file, and a file description. Consider the statement

```
open file(subscriber) title("vfile_ grp>reg") keyed sequential update;
```

In this statement, the file value is given by the file constant name 'subscriber', the title is the Multics attach description 'vfile_ grp>reg', and the file description is 'keyed sequential update'. The statement is interpreted as follows:

- If the associated I/O switch is not attached, the attach description in the 'title' option is used to perform the attachment.
- If the associated I/O switch is not open, the file description is used to establish its opening mode. If the switch is already open, its opening mode is checked to see if it is compatible with the file description. For this statement, the Multics file must be indexed and must be available for both reading and writing.
- The current record indicator and the next record indicator are set to the first record of the data set.
- Finally, the file-state block is marked 'open'.

When a data set is opened for 'output', the effect is to create a new data set unless the 'title' option specifies an '-extend' attachment. For example, the statement

```
open file(subscriber) title("vfile_ grp>reg") keyed sequential output;
```

has quite a different effect than the previous example. This statement deletes the data set designated by 'vfile_ grp>ref' and creates a new, empty data set whose organization conforms to the file description.

The 'title' option can be omitted from an 'open' statement. If the 'title' option is omitted, a default attach description is formed. The default attach description specifies the input/output module 'vfile_' and the system resident file designated by the file name. For example, the statement

```
open file(subscriber) keyed direct output;
```

is equivalent to

```
open file(subscriber) title("vfile_ subscriber") keyed direct output;
```

FILE DESCRIPTIONS

The following diagram gives every complete file description that can be used to open a data set for record input/output:

$$\left\{ \begin{array}{l} \text{keyed direct} \\ \text{sequential} \\ \text{keyed sequential} \end{array} \right\} \left\{ \begin{array}{l} \text{output} \\ \text{input} \\ \text{update} \end{array} \right\} [\text{environment(string value)}] \text{ record}$$

A specific file description consists of one of the three lines in the first pair of braces, followed by one of the three lines in the second pair of braces, followed by an optional 'environment(stringvalue)', followed by the 'record' attribute. The latter is enclosed in brackets to show that it can be omitted. There are other rules for shortening a file description, but they are complicated and their use is not recommended.

During the time a data set is open under a given file description, the attributes in that file description determine which input/output operations are permitted. The attributes with which the file description begins determine whether the operations can be keyed, sequential, or both, as follows:

keyed direct	permits keyed operations only
sequential	permits sequential operations only
keyed sequential	permits both keyed and sequential operations

The attribute with which the file description continues determines the kind of statement that can be used to perform input/output, as follows:

output	permits the use of a 'write' or 'locate' statement only
input	permits the use of a 'read' statement only
update	permits the use of a 'write', 'read', 'delete', or 'rewrite' statement only; however, a 'write' statement is permitted only if the file description includes the 'keyed' attribute

observe that 'update' permits almost any input/output statement; however, it does not permit the use of a 'locate' statement (which is rarely used in any case) or the use of a 'write' statement for a data set that is opened as unkeyed 'sequential'.

The 'environment(stringvalue)' attribute is used to read into or write from a varying string. On reading, the length of the record determines the length of the varying string, and on writing, the length of the varying string determines the length of the record. The length field of the varying string and unused space in the string are not written and, in this way, space is conserved in the file.

'close' Statement

The close statement has a simple form, as indicated by the following example:

```
close file(subscriber);
```

This statement marks the file-state block 'subscriber' closed. In addition, it clears and frees any buffers which have been allocated and set by previous based input/output operations. These buffers are discussed under "Based Input/Output", later in this section.

KEYED INPUT/OUTPUT OPERATIONS

When an input/output statement contains a 'key' or a 'keyfrom' option, it performs keyed input/output. The file on which such a statement operates can usually be either 'direct' or 'sequential', but it must be 'keyed' in any case.

Keyed 'write' Statement

Consider the statement:

```
write file(employee) keyfrom(ssno) from(item(3));
```

The file 'employee' must be a 'keyed output' or 'keyed update' file. The statement attempts to create a new record in the file 'employee' under the key given by the value of the character-string variable named 'ssno'. If the file is 'keyed sequential output', the key given by 'ssno' must be greater than any key already in the file (so the record goes at the end of the file); otherwise, the 'key' condition occurs. The created record becomes the current record and its value is the current value of 'item(3)'. However, the operation fails and the 'key(employee)' condition occurs if there is already a record in 'employee' under the key given by 'ssno'.

There are two ways to create a keyed sequential data set. The efficient way is to write the records in the order of ascending keys; and to achieve this, the programmer opens the data set as 'keyed sequential output'. The less restricted way is to write the records in any order and let Multics sort them out; and to do this, the programmer opens the data set as 'direct output'.

Keyed 'read' Statement

The 'read' statement with the 'key' option is used for keyed input from a file. Consider the statement:

```
read file(employee) key(ssno) into(rec.main);
```

The file 'employee' must be a 'keyed input' or 'keyed update' file. The statement attempts to find a record in the file 'employee' that has the key given by 'ssno'. If such a record is found, it becomes the current record and is read into the PL/I storage designated by 'rec.main'. The operation fails and the 'key(employee)' condition occurs if there is no record in 'employee' under the key given by 'ssno'.

The key associated with a record is a data field in the file outside the record, although it can, of course, be duplicated within the record.

Input and output values must be matched exactly. Suppose the following statements are executed in sequence:

```
write file(employee) keyfrom(ssno) from(item(3));
```

```
read file(employee) key(ssno) into(rec.main);
```

If 'item(3)' and 'rec.main' have exactly the same structural attributes, these statements are equivalent to:

```
write file(employee) keyfrom(ssno) from(item(3));
```

```
rec.main = item(3);
```

The appearance of an assignment statement is natural, since a value is being transmitted, by way of a record, from one variable to another. However, the equivalence just given breaks down when the structural attributes of 'item(3)' and 'rec.main' are not exactly the same. When data types do not match, PL/I does not convert the value, and when aggregate types do not match PL/I does not attempt to promote. Instead, any disagreement of structural attributes is an error.

Keyed 'delete' Statement

The 'delete' statement with the 'key' option is used for the keyed deletion of an existing record from a file. Consider the statement:

```
delete file(employee) key(ssno);
```

The file 'employee' must be a 'keyed update' file. This statement attempts to delete a record from the file 'employee' under the key given by 'ssno'. The key is deleted from the file as well as the record, so the key is unused in this file after the delete operation. The 'key(employee)' condition occurs if there is no record in 'employee' with the key given by 'ssno'. If the file opening is sequential, the current record indicator and next record indicator are both set to the record following the deleted record.

Keyed 'rewrite' Statement

The 'rewrite' statement with the 'key' option is used to write a new version of an existing record in a keyed file. Consider the statement:

```
rewrite file(employee) key(ssno) from(correction);
```

The file must be a 'keyed update' file. This statement attempts to output a record to the file 'employee' and enter it under the key given by 'ssno'. The new value of the record is taken from the variable 'correction', and the old value of the record is destroyed. The 'key(employee)' condition occurs if there is no record in 'employee' under the key given by 'ssno'.

Since the 'write' statement uses the 'keyfrom' option to specify the key, the programmer may be tempted to use a 'keyfrom' option in the 'rewrite' statement; but this is a syntactic error. In PL/I, the 'key' option is used when a statement attempts to find a given key in a file (as in the 'read', 'delete', and 'rewrite' statements), and the 'keyfrom' option is used when a statement attempts to introduce a given key into the file (as in the 'write' statement).

Example of Keyed Input/Output

Suppose a simple list of subscribers to a monthly magazine is stored as a keyed sequential file. Each record gives the name and address of a subscriber and the date of expiration of his subscription. The key for each record is a 12-character string which is made up of the zip code and other identifying information. The problem is to extend the date of expiration of subscribers as their subscriptions are renewed. The program is as follows:

```
RENEW:  proc;
        dcl sysin file;
        dcl given file;
        dcl skey char(12) var;
        dcl 01 subs,
            02 name char(30) var,
            02 address char(60) var,
            02 expire,
            03 month dec(2),
            03 year dec(2);
        open file(given) direct update;
        do while ("1"b);
            get list(skey);
            if skey = "END"
                then do; close file(given); return; end;
            read file(given) key(skey) into(subs);
            year = year+1;
            rewrite file(given) key(skey) from(subs);
        end;
    end;
```

An example of input for the program is:

```
"94305MARSA82"
"02139STEIS95"
"20742MARTB61"
"END"
```

This procedure reads keys from the input stream 'sysin' and adds 1 to the year of expiration for the corresponding record. The fact that the subscription file is declared to be 'direct' does not imply that the data set is not sequential; it only means that this use of the data set will not depend on whether or not it is sequential. Indeed, a later example in this section uses this same file for sequential input/output.

SEQUENTIAL INPUT/OUTPUT OPERATIONS

When an input/output statement does not contain a 'key' or a 'keyfrom' option, it performs sequential input/output. The file on which such a statement operates must have the 'sequential' attribute.

Sequential 'write' Statement

Consider the statement:

```
write file(subscriber) from(cust);
```

The file 'subscriber' must be an unkeyed 'sequential output' file. The statement creates a new record at the end of the file, and the current value of 'cust' is assigned to the record.

Sequential 'read' Statement

The 'read' statement without the 'key' option is used for sequential input from a file. Consider the statements:

```
read file(subscriber) into(cust);
```

and

```
read file(employee) keyto(ssno) into(cust);
```

The file 'subscriber' must be a 'sequential input' or 'sequential update' file (keyed or not), and the file 'employee' must be a 'keyed sequential input' or 'keyed sequential update' file. The indicator associated with the file is moved to the next record; that is, the current record indicator in the file-state block is given the value of the next record indicator, which is then advanced one record. Then the value of the new current record is assigned to 'cust' in PL/I storage. The 'keyto(ssno)' option causes the key associated with the current record to be assigned to 'ssno' in PL/I storage.

The 'read' statement can also be used to skip over records in a sequential file. Consider the statement:

```
read file(subscriber) ignore(3);
```

The file 'subscriber' must be a 'sequential input' or 'sequential update' file (keyed or not). If the next record indicator designates the *i*th record of the file, then 'ignore(3)' moves the indicator to the (*i*+3)th record. The current record indicator is then given the same value as the next record indicator. Thus a subsequent sequential read, rewrite, or delete will reference this record. If the end of the file is reached before the operation is complete, the current record indicator is set to the null record and the 'endfile(subscriber)' condition is signalled. The argument of the 'ignore' option must be greater than zero.

Sequential 'delete' Statement

The 'delete' statement without the 'key' option is used for deletion of the current record of the file. Consider the statement:

```
delete file(subscriber);
```

The file 'subscriber' must be a 'sequential update' file (keyed or not). The statement causes the current record to be deleted. The current record indicator and next record indicator are both set to the following record.

Sequential 'rewrite' Statement

The 'rewrite' statement without the 'key' option is used to write a new version of an existing record in a sequential file. Consider the statement:

```
rewrite file(subscriber) from(renewal);
```

The file must be 'sequential update' (keyed or not). The statement replaces the contents of the current record with the value of 'renewal' in PL/I storage; and the old value of the record is destroyed. If the file is not 'keyed', then the replacement value must have the same storage type as the former value of the record.

Example of Sequential Input/Output

Once again a list of subscribers to a monthly magazine is stored as a keyed sequential file. The problem is to read through the file sequentially, checking each record in turn to see if the subscription has run out. Those records that represent expired subscriptions are copied into another keyed sequential file. The program is as follows:

```
TARDY:  proc;
        dcl given file;
        dcl tardy file;
        dcl skey char(12);
        dcl 01 subs,
            02 name char(30) var,
            02 address char(60) var,
            02 expire,
            03 month dec(2),
            03 year dec(2);
        open file(given) keyed sequential input;
        open file(tardy) keyed sequential output;
        on endfile(given) goto EXIT;
        do while("1"b);
            read file(given) keyto(skey) into(subs);
            if 12*year+month < 12*74+3
            then write file(tardy) keyfrom(skey) from(subs);
            end;
EXIT:   close file(tardy);
        close file(given);
        end;
```

The program tests for subscriptions that expired before March 74. Its most interesting point, however, is the use of 'keyed sequential' output. These attributes require that 'tardy' be written in order of ascending keys; and this requirement is satisfied, since the program is copying from a file, 'given', that is 'keyed sequential' and necessarily satisfies the requirement. The program would still be valid if 'tardy' were declared 'keyed direct', but the useful fact that the records will be written in order of ascending keys would not be made explicit and PL/I might not perform the output as efficiently as possible. Therefore the declaration 'keyed sequential' is best.

BASED INPUT/OUTPUT OPERATIONS

Based input/output is a rather advanced and difficult technique of PL/I programming. Fortunately, based output is not important in Multics PL/I and need be given no more than passing mention at the end of this discussion. However, based input is useful, especially in commercial programming.

Consider a programming application in which the following vicious circle arises:

The records of a given input file are in several different structural forms, and therefore a particular record cannot be read into PL/I storage until the storage type of its values have been determined. However, the records occur in an unpredictable order, and the only indication of the storage type of a particular record is a code that is contained within the record. In short, the record cannot be read until its form is known and its form cannot be known until the record has been read.

The based input statement breaks this circle by reading a record into system storage and thus using a special technique not otherwise available to the user.

Based Input

A based input statement can be obtained by writing a 'read' statement with a 'set' option instead of an 'into' option. Three forms are possible, as the following examples show:

```
read file(log) key(itemno) set(ptr);  
read file(log) set(ptr);  
read file(log) keyto(itemno) set(ptr);
```

The argument of the 'set' option must be a target for a 'pointer' value. In each of these statements, the record is located just as it would be for the statement with an 'into' option; the first statement is a keyed operation, the second and third are sequential operations. When the record has been located, PL/I allocates enough storage from system storage to hold the value of the record, copies the record into the storage, and sets 'ptr' to point to the beginning of the allocated storage.

It is useful to think of the input process as follows: PL/I examines the record, allocates storage with exactly the same structural attributes as the record, and then reads the record into that storage. This could not be done with an ordinary 'read' statement (with an 'into' option); there is no way that a program can examine a record before reading it in, and an ordinary 'read' statement must specify the attributes of the target before the input is performed.

Once the record has been input by means of a based input statement, it is interpreted by using the pointer value to associate a based variable with the value of the record. Many techniques for the use of based variables can be used; but the most important ones are given in the following examples.

EXAMPLES OF BASED INPUT

In the first example, a file that contains the daily transactions of a repair shop must be read. There are different kinds of transactions, and therefore different kinds of structures are required to represent them. The transactions covered are as follows;

<u>Code</u>	<u>Purpose</u>
1	Order a replacement part, giving the part number and the source of supply.
2	Bill a customer, giving name, address, and amount due.
3	Record an internal charge, giving a cost center and a cost.

A program is required to read through the file accumulating the credits. A Code 1 transaction is ignored, but each Code 2 or Code 3 transaction contributes its 'amount_due' or 'cost' to the accumulated credits. When the file has been completely read, the accumulated total is printed.

If based input were not available, the file could be designed with two records for each transaction. The first record of any pair would give the transaction code and the second would be a structure describing the transaction. The program would read the code, choose the appropriate target for the second record, and then read the second record. The disadvantage of this approach is that it uses twice as many records as necessary, and in many cases, would nearly double the cost of both the storage and the processing.

Since based input is available, the file can be written with one record for each transaction. Each record is a structure whose first member is the transaction code and whose remaining members are appropriate to the kind of transaction described. When a record has been read, the code is examined (without looking at the rest of the record) and is used to select a statement that will use an appropriate based variable to interpret the value of the record. The program is as follows:

```
SUM:  proc;
      dcl total pic"$$$$$.v99";
      dcl trans file;
      dcl sysprint file;
      dcl P pointer;
      dcl 01 order based(P),
           02 code dec(1),
           02 part_number char(12),
           02 supplier dec(3);
      dcl 01 bill based(P),
           02 code dec(1),
           02 customer,
           03 name char(20) var,
           03 address char(40) var,
           02 amount_due pic"$$$$.v99";
      dcl 01 charge based(P),
           02 code dec(1),
           02 cost_center char(3),
           02 cost pic"$$$$.v99";
      on endfile(trans) goto EXIT;
      open file(trans) sequential input;
      total = 0;
```

```

LOOP:    read file(trans) set(P);
         goto L(order.code);
L(1):    goto LOOP;
L(2):    total = total + amount_due;
         goto LOOP;
L(3):    total = total + cost;
         goto LOOP;
EXIT:    put skip list("total billing: ", total);
         close file(trans);
         end;

```

This program is easy to read, but it is not so easy to write. The application of based variables to the input value must be programmed carefully because errors may not be detected.

The declarations of the structures 'order', 'bill', and 'charge' are as might be expected from the definition of the problem. The association of the pointer 'P' with each of the three structures saves writing later. For example, 'order.code' means 'P->order.code' because 'order' is declared 'based(P)'.

The based input statement at 'LOOP:' reads the value of a record into system storage and sets 'P' to the beginning of the value. The important statement is:

```

goto L(order.code);

```

This statement evaluates the transaction code by overlaying the structure 'order' on the input value and then obtaining the value of 'order.code'. The value of 'order.code' must be 1, 2, or 3 and the appropriate transfer to 'L(1)', 'L(2)', or 'L(3)' is performed.

Once the code has been examined, the correct choice for a based variable can be made. For example, at 'L(2)', it is known that the record represents a bill for a customer, and therefore reference can be made to 'amount_due'. The fully-qualified equivalent of this reference is 'P->bill.amount_due', and it means: treat the value pointed to by 'P' as if it were like the structure 'bill' and get the value of 'amount_due' from that structure.

In this second example, a program generates a two-dimensional array and writes it out as a one-record file. Later, another program reads the array and processes it. It is assumed that the programs are used repeatedly in this sequence, and that the array may have different extents from one application to the next; therefore, the extents cannot be given as constants. For these programs, self-describing structures are used; that is, each array is incorporated in a structure that contains integers that represent the bounds of the array. The self-describing structure is designed in a special way so that not only the programmer knows where the extents are stored, but the PL/I processor also knows. The program that outputs the array is, in part, as follows:

```
BUILD:  proc;
        dcl (m,n) fixed;
        dcl 01 S based(p),
            02 extents,
            03 e1 fixed,
            03 e2 fixed,
            02 A(m refer(e1):n refer(e2));
        dcl p ptr;
        dcl X file;

        ... (set m and n to the desired values) ...

        allocate S;

        ... (set the m*n values of the array) ...

        open file(X) sequential output;
        write file(X) from(S);
        close file(X);
        end;
```

The interesting action in this program is the 'allocate' statement. This statement allocates the structure 'S' in system storage using the current values of 'm' and 'n' as the bounds for the array. At the same time, the "allocate" statement assigns the bounds to 'e1' and 'e2', so that an explicit record of the bounds of the array is made. These are useful when the array is read in by the second program that follows:

```
USE:    proc;
        dcl (m, n) fixed;
        dcl P pointer;
        dcl 01 S based(P),
            02 extents,
            03 e1 fixed,
            03 e2 fixed,
            02 A(m refer(e1), n refer(e2));
        dcl X file;
        open file(X) sequential input;
        read file(X) set(P);
        close file(X);

        ... (make use of A)

        end;
```

Because the array is self-describing and uses 'refer' options in the proper way, PL/I "understands" that the extents of the array are given by 'e1' and 'e2'. Incidentally, the variables 'm' and 'n' are not used at all in 'USE'; they are given to cover the allocation of the array 'A', but the array is not allocated in this program. PL/I allocates storage according to the requirements of the input record, not the declaration of 'S'; but then a reference to 'A' is interpreted as if that storage had been allocated by 'allocate S;'.

Based Output

An ordinary output statement transfers a given value to a portion of system storage called an output buffer; then the actual output is performed under control of the operating system from that output buffer. There are certain techniques that permit a programmer to gain access to the output buffer, and these are described in the following paragraphs. These techniques should be used only when the need is clearly evident.

OMISSION OF THE 'from' OPTION

It is possible to perform output directly from the input buffer; that is, under certain circumstances, the 'from' option can be omitted from a 'rewrite' statement, as follows:

```
rewrite file(employee) key(ssno);
```

or

```
rewrite file(subscriber);
```

Such a statement is allowed only if the last input operation on 'employee' or 'subscriber', respectively, was a based 'read' operation. In that case, the 'rewrite' statement takes its output value from the input buffer associated with the file by the preceding 'read' statement. Thus it is possible to read the value of a record into a buffer, modify it in that buffer, and write the value out from the same buffer.

'locate' STATEMENT

A second facility for programmer control of buffers is the 'locate' statement. The 'locate' statement is related to the 'write' statement as the based 'read' statement is related to an ordinary 'read'. Consider the statement:

```
locate buf set(ptr) file(employee) keyfrom(ssno);
```

The 'buf' must be a 'based' variable, and if it is declared 'based(ptr)' then the 'set' option can be omitted. If the file is not keyed, the 'keyfrom' option must be omitted. The effect of this locate statement can be described in terms of replacement by two other statements. The 'locate' statement itself can be replaced by a statement to allocate 'buf', as follows:

```
allocate buf set(ptr);
```

Then, at a later point in the program, just before the next output operation on the file 'employee' or just before the closing of the file 'employee', the actions implied by the following statement are carried out:

```
write file(employee) keyfrom(ssno) from(buf);
```

Thus the 'locate' statement sets up an output buffer but the contents of the buffer are not written out until the last possible moment.

SPECIAL FEATURES

The record I/O operations of PL/I are designed to be used with record data sets in which the record lengths are known ahead of time (e.g., are constant length) or in which the records are self defining (e.g., contain the record length in the first word of each record). The following paragraph describes a Multics PL/I feature that permits record I/O operations to be used with data sets containing records of arbitrary length.

Environment (stringvalue)

This attribute is used in the declaration of a file constant or in an open statement to modify the normal interpretation of record I/O for varying strings. When a file state block has this attribute, a 'write from' statement whose source variable is a varying string writes out a record containing just the current value of the varying string, not a complete image of its storage. Similarly, a 'read into' statement whose target is a varying string sets the current value of the target to be equal to the contents of the record. The operation will be successful as long as the record length does not exceed the length of the string. Here is an example:

```
dcl x char(200) varying,  
    y char(100) varying;  
  
x = "abc";  
rewrite file(f) key("alpha") from (x);  
read file(f) key("alpha") into (y);
```

If the file f has the 'environment (stringvalue)' option, the 'rewrite' statement places a three character record in the file, and the 'read' statement sets the value of y to be 'abc'. If it does not have this attribute, the 'rewrite' statement places a 200-character record into the file, and the 'read' statement raises the record condition, because the length of y's storage is 100 characters.

*

CONDITIONS FOR RECORD INPUT/OUTPUT

In this discussion, the conditions that occur during record input/output are described. They are:

```
endfile(ref)  
key(ref)  
record(ref)  
transmit(ref)  
undefinedfile(ref)
```

where ref is a reference that yields a file value. The general rules for the use of conditions are given earlier, in Section XIII, "Condition Handling"; only a summary is given here.

As indicated above, each condition is defined separately for each file constant, and thus for each file-state block, whether it is open or not. The identifier 'endfile' is not a valid condition; but if 'subscriber' is declared 'file', then 'endfile(subscriber)' is valid.

As an example of condition handling, consider the statement:

```
on endfile(subscriber) goto EXIT;
```

When this statement is executed, it establishes the 'on' unit 'goto EXIT;' for the condition 'endfile(subscriber)'. If the 'endfile(subscriber)' condition is signalled, the 'on' unit itself is executed. When the block that contains the 'on' statement is deactivated, the 'on' unit is reverted and no longer responds to a signal.

When a condition occurs, PL/I takes either of two actions, as follows:

- If an 'on' unit is established for the condition, then that 'on' unit is executed. If the execution of the 'on' unit runs to completion, control goes back to the point in the program at which the interruption occurred, and execution is resumed in a reasonable way (depending on the particular needs of the condition).
- If no 'on' unit is established for the condition, then the default 'on' unit is executed. The default 'on' unit for each condition is described earlier, in Section XIII, "Condition Handling."

PL/I saves certain values before signalling a condition. For each kind of value saved, there is a stack and a built-in function. Just before the condition is signalled, the value is placed on the top of the stack, and after completion of the established 'on' unit, it is removed. The built-in function is used to access the value during the execution of the 'on' unit.

When a record input/output operation causes a condition to be signalled, the following values are saved in the manner just described. First, the file name, expressed as a character-string value, is saved in the stack associated with the 'onfile()' built-in function. Second, if the file being operated on has been opened with the attribute 'keyed', then the current key is saved in the stack associated with the 'onkey()' built-in function.

'endfile' Condition

Suppose the file 'subscriber' is positioned so that its current record is the last record in the file, and suppose a sequential 'read' statement is executed. Since there is no next record, PL/I signals 'endfile(subscriber)'. If an 'on' unit is established for 'endfile(subscriber)', then it is executed; and if the 'on' unit runs to completion, execution of the program resumes with the statement after the 'read' statement that caused the condition to occur.

Sometimes the number of records in a file is known in advance, and that number can be used to control the loop that reads the records. In such a case, an 'endfile' condition indicates an error in the preparation of the input file. The programmer may choose to provide an 'on' unit for recovery from such an error or he may decide to accept the diagnostic message and program abort that the system supplies by default.

An occurrence of an 'endfile' condition is not necessarily an error; indeed, it is an excellent way to terminate a loop that processes the records of a file. Several of the example programs given in this section use a statement such as the following:

```
on endfile(subscriber) goto EXIT;
```

to exit from a loop that is reading the records of an input file. Such programming is especially elegant. A programmer writes a loop that would go on reading records forever if there were no end to the file. Then quite separately, the programmer writes an 'on' statement that determines the action to be taken when the end of the file is reached. This separation of activities makes the program easier to write and easier to understand.

'key' Condition

The 'key(subscriber)' condition occurs when a wrong assumption is made about the keys in the file 'subscriber'. That is, it occurs when a keyed 'write' statement has a 'keyfrom' option which supplies a key which is already in the file; and it occurs when a keyed 'read', 'delete', or 'rewrite' statement has a 'key' option that supplies a key that is not already in the file. If an 'on' unit is established for the condition 'key(subscriber)' it is executed; and if it runs to completion, then execution continues with the statement after the input/output statement in which the condition occurred.

This condition has an important role in signalling errors in the use of a keyed file. It can also be used to support a legitimate inquiry about the use of a key in a file. For example, suppose a file of employees is keyed by social security numbers. Then a given number can be checked to see if its owner is an employee. First, the statement

```
on key(employee) emp = "0"b;
```

is executed. Then the following statements are used to branch according to whether or not the number was in the file:

```
emp = "1"b;  
read file(employee) key(given_ssno) into(info);  
if emp then goto EMPLOYEE; else goto NOT_EMPLOYEE;
```

'record' Condition

The 'record(subscriber)' condition occurs when the value of a record from the file 'subscriber' does not fit into the storage provided by the 'into' option of a 'read' statement. A value fits if the number of bits it requires is exactly the number of machine-level bytes allocated in storage. This condition is implementation dependent; however, the following assertion is true for any implementation of PL/I:

If the 'record' condition occurs, then the input value does not have the same storage type as the target given by the 'into' option of the 'read' statement.

Such a condition indicates an error in the program or the input file. After an established 'on' unit is executed, PL/I completes its execution of the 'read' statement by assigning the value of the record to the target, truncating it or padding it with zero-valued bits to make it fit the target.

Although it is an error to input a value that does not have the same storage type as the target, this condition only occurs when the number of bytes required by the value of the record differs from the number of bytes in the target. Thus, certain errors go undetected.

'transmit' Condition

The 'transmit(subscriber)' condition occurs when data cannot be reliably transmitted between the file 'subscriber' and the PL/I storage referenced in the statement that attempted the input or output. After an established 'on' unit is executed, the program resumes at the point following the input/output statement that caused the condition; but the value of the data transmitted by the statement is undefined.

The condition is usually caused by factors beyond the programmer's control, such as hardware failure, so the recovery procedure usually cannot be initiated until the hardware is repaired.

'undefinedfile' Condition

The 'undefinedfile(subscriber)' condition occurs when an 'open' statement attempts unsuccessfully to open the file 'subscriber'. The condition can occur when, for example, an attempt is made to open an unkeyed data set for 'keyed' input or output, or when, for another example, the 'title' option specifies an illegal attachment. After an established 'on' unit is executed, the program resumes at the point following the 'open' clause. This point may be the next of a series of 'open' clauses in an 'open' statement or the statement after the 'open' statement.

An occurrence of an 'endfile' condition is not necessarily an error; indeed, it is an excellent way to terminate a loop that processes the records of a file. Several of the example programs given in this section use a statement such as the following:

```
on endfile(subscriber) goto EXIT;
```

to exit from a loop that is reading the records of an input file. Such programming is especially elegant. A programmer writes a loop that would go on reading records forever if there were no end to the file. Then quite separately, the programmer writes an 'on' statement that determines the action to be taken when the end of the file is reached. This separation of activities makes the program easier to write and easier to understand.

'key' Condition

The 'key(subscriber)' condition occurs when a wrong assumption is made about the keys in the file 'subscriber'. That is, it occurs when a keyed 'write' statement has a 'keyfrom' option which supplies a key which is already in the file; and it occurs when a keyed 'read', 'delete', or 'rewrite' statement has a 'key' option that supplies a key that is not already in the file. If an 'on' unit is established for the condition 'key(subscriber)' it is executed; and if it runs to completion, then execution continues with the statement after the input/output statement in which the condition occurred.

This condition has an important role in signalling errors in the use of a keyed file. It can also be used to support a legitimate inquiry about the use of a key in a file. For example, suppose a file of employees is keyed by social security numbers. Then a given number can be checked to see if its owner is an employee. First, the statement

```
on key(employee) emp = "0"b;
```

is executed. Then the following statements are used to branch according to whether or not the number was in the file:

```
emp = "1"b;  
read file(employee) key(given_ssno) into(info);  
if emp then goto EMPLOYEE; else goto NOT_EMPLOYEE;
```

'record' Condition

The 'record(subscriber)' condition occurs when the value of a record from the file 'subscriber' does not fit into the storage provided by the 'into' option of a 'read' statement. A value fits if the number of bits it requires is exactly the number of machine-level bytes allocated in storage. This condition is implementation dependent; however, the following assertion is true for any implementation of PL/I:

If the 'record' condition occurs, then the input value does not have the same storage type as the target given by the 'into' option of the 'read' statement.

Such a condition indicates an error in the program or the input file. After an established 'on' unit is executed, PL/I completes its execution of the 'read' statement by assigning the value of the record to the target, truncating it or padding it with zero-valued bits to make it fit the target.

Although it is an error to input a value that does not have the same storage type as the target, this condition only occurs when the number of bytes required by the value of the record differs from the number of bytes in the target. Thus, certain errors go undetected.

'transmit' Condition

The 'transmit(subscriber)' condition occurs when data cannot be reliably transmitted between the file 'subscriber' and the PL/I storage referenced in the statement that attempted the input or output. After an established 'on' unit is executed, the program resumes at the point following the input/output statement that caused the condition; but the value of the data transmitted by the statement is undefined.

The condition is usually caused by factors beyond the programmer's control, such as hardware failure, so the recovery procedure usually cannot be initiated until the hardware is repaired.

'undefinedfile' Condition

The 'undefinedfile(subscriber)' condition occurs when an 'open' statement attempts unsuccessfully to open the file 'subscriber'. The condition can occur when, for example, an attempt is made to open an unkeyed data set for 'keyed' input or output, or when, for another example, the 'title' option specifies an illegal attachment. After an established 'on' unit is executed, the program resumes at the point following the 'open' clause. This point may be the next of a series of 'open' clauses in an 'open' statement or the statement after the 'open' statement.

SECTION XVI

PL/I IN THE MULTICS SYSTEM

Both the PL/I compiler and the programs it compiles execute in the Multics system. To compile and execute PL/I programs, the user must understand the fundamentals of Multics. This section discusses those aspects of Multics that are of particular interest to a PL/I programmer. Other manuals describe Multics in greater detail; the Multics Users' Guide provides an introduction to Multics and the Multics Programmers' Manual gives a detailed description.

This section has three parts. The first part is a brief description of the Multics storage system. The second part describes the facilities of Multics that are used for compiling and executing PL/I programs, including the PL/I compiler itself, the mechanism for linking external procedures, the relation of a program to a Multics process, and the attachment of files. The third part gives the procedure for running a PL/I program in Multics and includes a complete example.

STORAGE SYSTEM SEGMENTS

The basic unit of information in the Multics storage system is the segment. The Multics storage system consists of two kinds of segments, namely: directory segments and non-directory segments. A directory segment contains a list of segment names and segment attributes. A nondirectory segment, or simply segment, contains data or code.

Names

Each segment has one or more absolute pathnames. An absolute pathname is a sequence of segment names starting from the root directory and proceeding through directory segments to the designated segment. A more convenient way to reference a segment is by its relative pathname. The relative pathname is a sequence of segment names starting from the user's current location in the storage system to the designated segment. The user's current location in the storage system is called the current working-directory. A working-directory, usually based on the user's name and project, is established by the system at log-in. For example, suppose a user logs in, as follows:

```
login Noman
password
```

As part of the log-in procedure, the system establishes a working-directory for the user Noman. The pathname for a typical working-directory might be:

```
>udd>ProjGiant>Noman
```

If the user then creates a segment 'alpha', the following pathnames apply:

```
absolute pathname >udd>ProjGiant>Noman>alpha
```

```
relative pathname alpha
```

A complete discussion of the log-in procedure and the naming conventions can be found in the Multics User's Guide.

Component Names

A segment name can consist of a sequence of one or more component names. The character '.' is used to separate one component name from another. For example, consider the following segment name:

```
alpha.beta.gamma
```

This segment name has three components, namely: 'alpha', 'beta', and 'gamma'. The Multics PL/I compiler requires that the name of the segment to be compiled end with the component '.pl1'. The procedure 'RANGE', for example, is entered as a source segment named 'RANGE.pl1'.

Entry Point Names

A location within a segment that is known externally by a symbolic name is called an entry point name. Such a location can be referenced by specifying the segment name and the entry point name in the following way:

```
sn$epn
```

where sn is the segment name and epn is the entry point name. When the segment name and the entry point name are the same, the reference can be abbreviated to simply the entry point name. For example, the entry point name 'z' in the segment 'y' is referred to as 'y\$z'. The entry point name 'z' in the segment 'z' is referred to as 'z\$z' or, in the abbreviated form, as 'z'.

Reference Names

The name that a process uses to refer to an external variable or procedure is called a reference name. Binding between a reference name and the object it references is determined by the binder or the dynamic linking facility.

MULTICS PL/I COMPILER

The Multics PL/I compiler produces an object segment and an optional listing segment. The names of these segments are derived from the name of the source segment, sn, as follows:

<u>Segment</u>	<u>Name</u>
source	sn.pl1
object	<u>sn</u>
listing	sn.list

where sn is the segment name specified by the user.

Entry Names

The segment name for an object segment is not necessarily the same as the entry point name. Consider the following procedure:

```
p:  proc;
    ...
x:  entry;
    ...
y:  entry;
    ...
    end;
```

This procedure has three entry point names: 'p', 'x', and 'y'. If the procedure is entered as a source segment named 'p.pl1', the object segment is named 'p'. The entry point names within the segment 'p' are as follows:

```
p$p
p$x
p$y
```

The first name can be abbreviated, according to the rules for entry point names, to simply 'p'. The other names, however, must be referred to as 'p\$x' and 'p\$y'.

To avoid confusion between entry point names and segment names, the programmer can name each source segment with the name of its major entry. If a procedure has multiple entry point names, the object segment can be given additional names by the use of the addname command. In this way, references that include the character '\$' can be avoided.

For example, in the procedure 'p' just given, the object segment 'p' can have the names 'x' and 'y' added to it.

Object Segment

If no fatal errors are encountered in a compilation, the PL/I a compiler produces a standard Multics object segment, which consists of the following sections:

<u>Section</u>	<u>Description</u>
text	the binary machine language program
definitions	the set of character string names of entry points to this segment and of any procedures called by this segment.
link	the prototype linkage section, to be copied into the linkage/static segment when the procedure is first referenced. PL/I internal static variables are allocated in this section.
symbol	the relocation bits for the text and linkage sections. This section is also used for the symbol table, if one is requested.

A complete description of a standard Multics object segment, including an alternative object segment layout that has a separate static section, is given in the Multics Programmers' Manual.

The `print_link_info` command can be used to obtain information about a given object segment.

Listing Segment

The listing segment contains the output listing produced by the PL/I compiler. This output listing is divided into five sections, as follows:

<u>Section</u>	<u>Description</u>
source	a line-numbered copy of the source segment
symbol	a table of the names declared in the program, the storage requirements, and a list of external names
error	a list of error messages
map	an object code map, which gives for each statement the location of the beginning of the instruction sequence in the object program
list	a listing of the object program in an assembly-like language

By specifying control arguments in the `pl1` command, the user can request one or more sections of the compiler output listing. If no control arguments are specified, no listing is produced.

LINKING

A PL/I program is defined as a set of external procedures and their operating environment. If a program contains more than one external procedure, cross references between the procedures must be resolved. Similarly, references to external variables must be resolved. In Multics this is normally done by a mechanism known as dynamic linking. For each external object referenced in a procedure the procedure's linkage section contains a link. This is a specially formatted double word that is used as a pointer value by the procedure's object code. The first time the link is referenced during the program's execution, a fault occurs. This fault causes a system routine called the linker to be invoked. The linker determines the pointer value that denotes the external object and replaces the link with this value. All subsequent references to the link get the correct pointer value without intervention of the linker.

The dynamic linking facility eliminates the need for a preliminary operation that links together all the procedures of a program. During program development this facility has many advantages. For example:

- A program can avoid the problems associated with the use of an obsolete version of a procedure. As soon as an error is detected and corrected, the new version is available to all users.
- A program can be tested before all the procedures it uses are available. Since the execution of the program can proceed until a missing procedure is invoked, useful debugging runs can be made.
- A program can include references to large special-purpose procedures that are called only under unusual circumstances without incurring unnecessary overhead.
- The target of a link can be temporarily changed, by the `initiate` command, to test a new procedure.

The following paragraphs give more information about dynamic linking and briefly describe an alternative facility known as binding. For full details on both topics consult the Multics Programmers' Manual, Reference Guide.

Search Mechanism

This section explains how the linker finds the target of a link.

For an ordinary external variable, the linker uses a system maintained table to find a location in a system storage pool. The location of a particular variable is assigned the first time a link to that variable is snapped. The table and storage pool are part of the user's process.

For other external objects, the linker uses a two step method. First, it finds the object segment that is supposed to contain the link's target, then it searches definitions in the object segment for the appropriate name. Normally the name resolves to a procedure entry point, but for external variable names of the form 'a\$b', the name resolves to a data location within the object segment or within its static storage. For example, the name `iox_$user_input`, resolves to a ptr in the static storage associated with the system routine `iox_`. Object segments containing such external variables are created by means other than the PL/I compiler.

To find the segment containing the target of a link, the linker uses the reference name component of the external name and a set of search rules. The normal search rules are as follows:

- 1) Search the list of initiated reference names and segments. The first time a link is snapped to a segment with a particular name, the name is placed on this list. Subsequent links to this name resolve to the associated segment. In the course of a process, several names may be initiated for the same segment.
- 2) Search the referencing directory for a segment whose name is the same as the reference name.
- 3) Search the working directory in the same fashion. At all times the system denotes some directory as the working directory for the user's process. The user can change the working directory by use of the `change_wdir` command.
- 4) In the same fashion, search a set of library directories.

The search stops as soon as a segment with the required name is found. If that segment does not contain the correct entry point, an error is signalled.

The `set_search_directory` and `set_search_rules` commands may be used to establish different search rules. See the Multics Programmer's Manual, Commands, for details.

Hidden Dangers of Dynamic Linking

The programmer who is unaware of the fundamental workings of the dynamic linking mechanism may experience unexpected difficulties when he attempts to execute programs in Multics. Most problems are caused by the fact that the system maintains the association between a segment name and its address throughout the life of the process. For example, if a programmer executes procedure A, which calls a library procedure X, and he then changes to a new working directory and executes procedure B which calls an external procedure X located in his new working directory, the system will establish a link to the original segment.

The same persistence of meaning occurs for external variables, and this is a more common source of trouble. Suppose, for example, that the programmer executes a program using external file F declared as:

```
dcl F file stream input;
```

and then executes a program in which F is declared as:

```
dcl F file record update;
```

He will get an error message to the effect that the file's attributes conflict with its usage. This is because the first program's use of F established it as a 'stream input' file.

The general problem here is that the user's program really consists of all PL/I procedures invoked in the process. Therefore, an external name can only be used for one purpose throughout the life of the process.

A general way to get around problems of this sort, is to use the new_proc command before running a new program that may conflict with the old.

Binding

When a program that consists of several external procedures is ready for production, a bound segment can be created for the program by the use of the Multics bind command. This command packs a group of separately compiled procedures into a single object segment in which links within the segment are permanently linked and multiple outward references to the same target are condensed into a single outbound link.

The creation and maintenance of the bound segment involves an additional step, but the execution of the program as a bound segment is efficient. In fact, the use of a bound segment has most of the advantages of compiling procedures together but avoids the problems of a large compilation. For example, if one of the procedures that make up a bound segment is found to contain an error, the procedure can be recompiled. Then, the procedures can be rebound using the corrected version of that compiled procedure.

INITIALIZATION AND ALLOCATION OF VARIABLES

The initialization and allocation of variables, file openings, and segment linking are related to the duration of a process, as follows:

- An 'internal static' variable declared with the 'initial' attribute is initialized the first time the linker snaps a link to its object segment. When the procedure is first invoked, the variable has the initial value. During this or subsequent invocations, the value of the variable may be explicitly changed; however, the value of the variable is not restored to its initial value at each subsequent procedure invocation. The value of such a variable is known throughout the life of a process.
- An external static variable is allocated as described under "Linking." If declared with the 'initial' attribute it is initialized when allocated. During this or subsequent procedure invocations, the value of the variable may be explicitly changed; however, the value of the variable is not restored to its initial value at each subsequent procedure invocation. The value of such a variable is known throughout the life of a process.
- A 'controlled' variable is allocated in system storage and remains allocated unless explicitly freed by the execution of a 'free' statement. A controlled variable has an associated control block that is allocated as though it were a static variable.
- A 'based' variable can be allocated in system storage or in an area or can be equivalenced to an already allocated variable. Such variables can be allocated in permanent or per-process temporary areas. If the variable is allocated in a permanent area, it is available from process to process. However, the value of a pointer, file, entry, or label variable is valid only for the life of the process in which it is set.
- A file opened in a process remains open for the duration of the process unless it is explicitly closed by the execution of a 'close' statement.

ATTACHING FILES

The attachment of a PL/I file to a Multics file during program execution was described earlier in Sections XIV and XV, "Stream Input/Output" and "Record Input/Output," respectively. This attachment can also be performed at command level by the `io_call` command. If an I/O switch is attached at command level, any attachment specified within the program for the associated file is ignored. Command level attachment allows a program to be device independent. A programmer can run the same program under different input/output environments by using different variations on the `io_call` command.

Consider, for example, the following program:

```
test:  proc;
       dcl (a,b) file;
       ...
       open file(a) title("vfile_alpha") stream input;
       open file(b) keyed sequential output;
       ...
       end;
```

If the I/O switches 'a' and 'b' are not attached when the program is executed, the attach description 'vfile_alpha' is used to attach switch 'a' and the attach description 'vfile_b' to attach switch 'b'. Input is then read from the storage system file 'alpha' and output is written to the storage system file 'b'. The user can specify a different source and target for these files by attaching the switches at command level. Consider the following sequence of commands:

```
io_call attach a syn_user_input
io_call attach b vfile_x
test
io_call detach a
io_call detach b
```

As a result of the attachments made by the `io_call` commands, when the program is executed, input is read from the standard input switch and output is written in the storage system file 'x'.

Note that if an I/O switch is attached at command level, it must also be detached at command level. Furthermore, it is an error to detach an I/O switch if the corresponding file-state block has not been closed. If there is any doubt about the status of a file-state block, the `file_status` command can be executed.

The following paragraphs describe the I/O switch, the `io_call` command, and the I/O modules that can be specified.

I/O Switch

An I/O switch contains the following items:

- The switch name. This value is a character string that identifies the switch. The switch name is the same as the PL/I file name.
- The control block pointer. This value is a pointer to the control block maintained by the I/O system.
- The attach description. This value is a character string that gives the I/O module and source or target for the input/output operations. The form of the attach description depends upon the I/O module.
- The opening mode. This value is a character string that describes the type of processing to be done.

When the switch is in the detached state, the attach description is an empty string. Similarly, when the switch is in the closed state, the open description is an empty string. A switch can be open only if it is attached.

STANDARD SWITCHES

As part of the standard initialization of a Multics process, the switch 'user_i/o' is attached to the user's terminal. The following switches are attached as synonyms for 'user_i/o':

```
user_input
user_output.
error_output
```

The attachment of the above three switches can be changed.

io call Command

The io_call command performs an operation on a designated I/O switch. The operations attach and detach are described here. A complete description of this command is given in the Multics Programmers' Manual.

To attach an I/O switch the following form is used:

```
io_call attach sn ad
```

where sn is the switchname and ad is the attach description.

The attach description for each of the most commonly used I/O modules is given later in this section.

To detach an I/O switch the following form is used:

```
io_call detach sn
```

where sn is the switchname.

The current attachment of all the I/O switches in a process can be obtained by the `print_attach_table (pat)` command.

I/O Modules

Some I/O modules that are relevant to PL/I stream and record input/output are described in the following paragraphs. Detailed descriptions of these and other I/O modules including modules for tape I/O can be found in the Multics Programmers' Manual.

vfile_ I/O MODULE

The vfile I/O module performs input from or output to a storage system file. The attach description for this I/O module has one of the following forms:

```
vfile_ pn
```

```
vfile_ pn -extend
```

where pn is the pathname of the segment.

The `-extend` control argument indicates that in an opening for 'output', information is to be added to the file; that is, the file is to be extended. When the file is opened for 'output', the pointer to the file is positioned to the end of the file.

If the `-extend` control argument is not specified and the file is opened for output, any existing information in the file is destroyed.

tty_ I/O MODULE

The tty_ I/O module performs input from or output to a terminal device. The attach description for this module has the form:

```
tty_ dn
```

where dn is a character string that identifies the device.

syn_ I/O MODULE

The `syn_` I/O module is used to attach a switch as a synonym for another switch. The attach description for this module has the following form:

```
syn_ sn
```

where sn is the name of the I/O switch whose attachment is to be used.

The I/O switch sn can itself be attached as a synonym for another switch. However, the I/O switch that is the final destination of the synonym attachment must be attached when the switch for which it is a synonym is opened.

The file description for any file-state block attached in this way must be compatible with the I/O module and file designated for the attached switch.

record_stream_ I/O MODULE

The `record_stream_` I/O module attaches a source switch to a target switch so that record operations on the source switch are converted to stream operations on the target switch or stream operations on the source switch are converted to record operations on the target switch. The attach description for this module has one of the following forms:

```
record_stream_ tsn  
record_stream_ tsn -nnl  
record_stream_ tsn -length n  
record_stream_ -target ad  
record_stream_ -nnl -target ad  
record_stream_ -length n -target ad
```

where tsn is the target switch name, n is the length, and ad is an attach description.

The `-nnl` control argument is used for record to stream conversion. If this control argument is not specified for record to stream conversion, a record is taken from the source switch, a newline character is appended, and the resulting string is given to the target switch. If the `-nnl` control argument is specified, the record is taken from the source switch and given to the target switch without modification.

The `-length n` control argument is used for stream to record conversion. If this control argument is not specified for stream to record conversion, a string of bytes ending with a newline character is taken from the source switch, the newline character is removed, and the resulting string is given to the target switch. If the `-length n` control argument is specified, a record is formed by taking n bytes from the source switch and giving these bytes to the target switch.

The target switch can be specified either by name or by attach description. If the target switch is specified by attach description, a unique name is created for the switch.

The `record_stream_` I/O module makes it possible for record input/output statements to process an unstructured file and for stream input/output statements to process some structured files.

As an example of the use of the `record_stream_` I/O module, consider the following program:

```
p:  proc;
    dcl (a,b) file;
    dcl calc entry(char(80));
    dcl endfile condition;
    dcl alpha char(80);
    on endfile(a) goto exit;
    open file(a) sequential input;
    open file(b) sequential output;
loop: read file(a) into(alpha);
      call calc(alpha);
      write file(b) from(alpha);
      goto loop;
exit: close file(a,b);
      end;
```

If the `record_stream_` I/O module is specified for the I/O switches 'a' and 'b', the record input/output of the program 'p' can be directed to a terminal. Consider the following sequence of commands:

```
io_call attach a record_stream_ user_input
io_call attach b record_stream_ user_output
p
...
io_call detach a
io_call detach b
```

The source switch 'a' is attached to the target switch 'user_input' and the source switch 'b' is attached to the target switch 'user_output'. Input is taken from the user's terminal and output is directed to the user's terminal. The characters '...' in the above command sequence indicate the input/output produced by the execution of 'p' on the user's terminal.

RUNNING A PL/I PROGRAM IN MULTICS

A PL/I program often consists of some new external procedures and some previously-compiled external procedures. Running such a program involves the following activities:

- Each new procedure is entered as an ASCII source segment.
- Each new procedure is compiled into an object segment using the Multics PL/I compiler.
- The program is executed by typing the entry name of a procedure as a Multics command. When a separately compiled external procedure is called by another procedure, it is linked dynamically.
- If the program does not execute properly, a Multics debugging tool is used to locate the source of the error.
- When the program is debugged, the program's performance can be measured and, in some cases, improved.

The following paragraphs discuss these activities.

Entering an External Procedure

To enter the source for an external procedure, the programmer uses an editor to create an ASCII source segment. Several general-purpose editors are available in Multics for the creation and editing of ASCII segments. The most commonly used editors are `edm` and `qedx`.

The standard Multics text editor, `edm`, is easy to learn and use. The `qedx` editor is more powerful than `edm` but harder to learn. In addition to the capabilities for interactive text modification, the `qedx` editor has a macro facility that can be used to construct editing programs for the systematic modification of text.

The Multics Introductory Users' Guide illustrates the use of the `edm` editor to enter a source segment. The `qedx` editor is illustrated later in this section for the same purpose. A detailed description of both of these editors can be found in the MPM Commands.

Compiling an External Procedure

To compile an external procedure, the user invokes the PL/I compiler by typing the `pl1` command. The first argument of the `pl1` command identifies the source segment to be compiled. If the compilation is successful, an object segment is usually produced.

`pl1` COMMAND

The `pl1` command has the following form:

```
pl1 path {-control_args}
```

where:

1. `path`
is the pathname of the PL/I source segment.
2. `control_args`
are one or more control arguments, separated by blanks. These arguments are optional.

For a detailed description of the `pl1` command, see the MPM Commands.

The following list gives the control arguments and abbreviations that can be specified in the second form. For each control argument a brief description is given.

<u>Control Argument</u>	<u>Meaning</u>
-check -ck	perform a syntactic and semantic check of the program, and omit the code generation.
-optimize -ot	optimize the efficiency of the object segment.
-brief -bf	produce the short form of the error message on 'user_output'.
-source -sc	produce the source section of the compiler output listing in the listing segment.
-symbols -sb	produce the source and symbols sections in the listing segment.
-map	produce the source, symbols, error, and map sections in the listing segment.
-list -ls	produce the complete compiler output listing (all five sections) in the listing segment.
-table -tb	generate a full symbol table for use by symbolic debugging routines.
-brief_table -bftb	generate a partial symbol table.
-profile -pf	generate additional code to meter the execution of individual statements.
-severity <i>i</i> -sv <i>i</i>	suppress error messages whose severity level is less than <i>i</i> ($1 \leq i \leq 4$).

Executing a Program

To execute a program, the user invokes a procedure by typing its entry name as a Multics command. The Multics command, in this case, has one of the following forms:

```
p
p a ...
```

where p designates the object segment and entry point as described earlier and a is an argument. The form 'a ...', in the second form, indicates that one or more arguments, separated by blanks, can be given.

A procedure that does not have any arguments is executed by typing the first form, as follows:

```
RANGE
```

Multics treats the external entry name as a command to locate and begin the execution of the segment RANGE containing the entry name RANGE\$RANGE.

A procedure that has only parameters declared 'char(*)' can be executed, using the second form. For example, consider the following external procedure:

```
p:  proc(x,y);
    dcl (x,y) char(*);
    ...
    end;
```

The procedure 'p' is executed as a Multics command, as follows:

```
p alpha 1
```

The Multics system interprets this line as the command 'p' with the arguments 'alpha' and '1'. This command provides arguments in the same way as the following call statement within a PL/I program:

```
call p("alpha","1");
```

When a procedure is invoked in this way, no check on the number of arguments is performed. If the correct number of arguments is not given, the results of the procedure's execution are undefined. To check the number of arguments, the cu_ command, described in the Multics Programmer's Manual, can be used.

PROGRAM TERMINATION

If a program executes correctly, it usually returns to command level when the processing is complete. If a program contains errors, its termination is unpredictable. Sometimes, the program completes processing but produces incorrect results. Sometimes, the user presses the interrupt key on his terminal to suspend its execution. Sometimes, the system suspends the program's execution due to the occurrence of an exceptional condition and returns to command level.

When a program is suspended, the user can terminate its execution by typing the command release, can continue its execution by typing the command start or can invoke another procedure. When a program is suspended during debugging, the user can invoke one of the Multics debugging tools to locate the source, provided that its suspension was not due to the occurrence of a fatal error.

Debugging a Program

To debug a program, the user can select one of the debugging tools provided by the Multics system. Two useful tools for debugging PL/I programs are the commands probe and trace. An introduction to these debugging aids is given in the following paragraphs. A detailed description can be found in the Multics Programmers' Manual.

probe COMMAND

The probe command invokes a debugging system that allows the user to interactively examine the state of his program. The probe command can be used for the following purposes:

- to look at or modify the value of a variable
- to set a breakpoint
- to examine the stack of block activations
- to invoke external subroutines and functions

The probe system operates in response to user requests. The user can, for example, isolate a program bug by setting breakpoints at strategic points within his program. When the execution of a program halts at a breakpoint, the user can examine the values of key variables. The execution of the program can continue in this way, from breakpoint to breakpoint, until the source of the problem is discovered.

The probe system requires a symbol table and statement map for symbolic debugging. These are produced when the table control argument is specified in the pl1 command.

trace COMMAND

The trace command invokes a tool for monitoring all calls to a specified set of external procedures. The trace command is useful for both debugging a program and measuring its performance. The trace command can be used for the following purposes:

- to print arguments on entry or exit
- to stop on entry or exit
- to specify the times when the trace occurs
- to execute a Multics command line on entry or exit
- to meter time spent in a procedure

The user can, for example, request that the arguments for a procedure be printed on entry to a procedure and, in this way, a trace of the calls on a procedure with the value of each of its arguments is produced.

Measuring a Program's Performance

The cost of executing each statement of a program can be determined by specifying the -profile control argument of the pl1 command. The information produced is of interest to both the beginning programmer and the expert. For the beginning programmer, it is a guide to the economics of programming and restores the view of hardware cost that a high-level language otherwise obscures. For the expert programmer, it is an indication of the points in a program that are unreasonably expensive and that require refinement.

To measure the performance of a program, the user specifies the -profile control argument in the pl1 command that compiles the external procedures of the program. When the -profile control argument is specified, additional code is generated to calculate statistics about the execution of each statement. After the program has been executed, the segment that contains the accumulated statistics can be examined by executing the profile command. For a further description of the -profile control argument and the profile command, see the MPM Commands.

For each statement in each line of the PL/I external procedure, a line is printed that gives the number of times the statement was executed, the number of instructions executed, and the support subroutines called as a result of the statement's execution. For example, consider the following lines from a profile listing:

LINE	STM	COUNT	COST	PROGRAM
8	1	1	7	
10	1	1	10 + 2	(stream_io put_end)
12	1	5	50 + 15	(stream_io put_list_al put_end)

The profile listing indicates that the statement on line 8 was executed once; this statement requires seven machine language instructions and does not require any support subroutines. The statement on line 10 was executed once; this statement requires ten machine language instructions and two support subroutines, namely stream_io and put_end. The statement on line 12 was executed five times; this statement requires ten machine language instructions and three support subroutines per execution.

EXAMPLE OF RUNNING PL/I

The external procedure 'RANGE', introduced earlier in Section XIV, "Stream Input/Output," is used here as an example of running a PL/I program in the Multics system. A script showing the entry, compilation, and execution of the program 'RANGE' is given in the following paragraphs.

The program 'RANGE' computes the range of an artillery piece fired on ground level. For each trajectory, 'v0' is the initial velocity, 'theta' is the angle of elevation, and the result, 'range', is the horizontal distance to impact.

Entering the Example

To enter the text of the PL/I source program at a terminal, the user calls one of the text editors available in the Multics system. For this example, qedx is used. The exclamation point is used to indicate lines typed by the user. The script begins as follows:

```

! qx
! a
! RANGE: proc();
!   dcl (v0, theta, range) float(15),
!       g float(15) init(32.174);
!   dcl sind builtin;
!   do while("1"b);
!       get data(v0, theta);
!       if v0=0 then return;
!       range = ((v0**2)*sind(2*theta))/g;
!       put skip data(v0, theta, range);
!       put skip;
!       end;
!   end RANGE;
! \f
! w RANGE.pl1
! q
r 1335 1.083 10.048 311

```

The log-in sequence is omitted from the script above. As the script begins, the user types the command `qx` to invoke the `qedx` text editor. The append request, `a`, is given to the editor and the editor awaits input. The user types the text of the program followed by `\f` to terminate the input. Finally, the user types the command `w` to write the text as a Multics file named `'RANGE.pl1'`.

The user is finished with the text editor, and he passes control back to the Multics command level by means of the quit request, `q`. The system responds with a ready message.

Compiling the Example

To compile the external procedure, the user invokes the PL/I compiler by the Multics command `pl1`. The segment just created is specified as the source and the control argument `'map'` is specified to produce a listing segment. The script continues as follows:

```
pl1 RANGE -map
PL/I

WARNING 75
The undeclared identifier "sysprint" has been contextually declared
as a file constant. It will acquire default attributes.

WARNING 75
"sysin"
r 1340 5.101 77.135 264
```

Diagnostic remarks are printed on the terminal by the compiler. In this case, a warning was issued, but the default interpretation is the intended interpretation. The programmer might elect to add:

```
dcl (sysprint, sysin) file;
```

to the program to eliminate the warning; but the program can be run as is. The diagnostic messages are explicit, English-language messages; therefore, no list of error messages is included in this manual.

The PL/I compiler produces an object segment that can be called from the programmer's terminal or from another external procedure.

Executing the Program

After the procedure is successfully compiled, it can be called at the Multics command level. The script continues with the execution of the procedure as follows:

```
RANGE
v0=1000 theta=35;

v0= 1.0000e+003   theta= 3.5000e+001   range=2.9207e+004;
v0=1000 theta=40;

v0= 1.0000e+003   theta= 4.0000e+001   range= 3.0609e+004;
v0=1000 theta=45;

v0= 1.2800e+003   theta= 4.5000e+001   range= 5.0923e+004;
v0=0 theta=45;
r1344 3.899 41.302 622
```

In order to execute the program, the user simply types the name of one of its external entries (there is only one in this case, 'RANGE'). The user then calculates four trajectories, supplying 'v0' and 'theta' each time and getting 'v0', 'theta', and 'range' back. The program is designed so that when the user enters a zero value for 'v0' the program terminates. At this point, the user proceeds to other computing activities or logs out.

Program Listing

In the script above, the pl1 command was given with the control argument -map. The result is the preparation of a listing segment that contains the source, symbol, error and map sections. The symbol section is an excellent guide to the declaration of identifiers in PL/I, and the beginning programmer should request it and study it for several of his programs. It shows how PL/I supplies missing attributes and it shows how identifiers are declared by context or implication as well as by 'declare' statements. The map section gives the address information usually expected from a storage map.

The listing is printed in a 132 characters/line format, which does not fit on this page, so it is compressed horizontally here. Otherwise, the listing for 'RANGE' is unchanged.

SOURCE LISTING

The source listing gives the version of the compiler used for the compilation, the time and date of the compilation, and any options specified. Then a line-numbered listing of the source program is given. The source listing for the example 'RANGE' is as follows:

COMPILATION LISTING OF SEGMENT RANGE
 Compiled by Multics PL/I Compiler of November 24, 1975.
 Compiled on : 03/09/76 0927.3 est Tue
 Options map

```

1 RANGE: proc();
2     dcl (v0, theta, range) float(15),
3         g float(15) init(32.174);
4     do while ("1"b);
5         get data(v0, theta);
6         if v0=0 then return;
7         range = (v0**2*sind(2*theta))/g;
8         put skip data(v0, theta, range);
9         put skip;
10        end;
11    end RANGE;
```

Symbol Listing

The symbol listing gives the names declared in the external procedure, the storage requirements, and the external names. The symbol listing for the example 'RANGE' is as follows:

NAMES DECLARED IN THIS COMPILATION.

IDENTIFIER	OFFSET	LOC	STORAGE	CLASS	DATA TYPE	ATTRIBUTES AND REFERENCES
NAMES DECLARED BY DECLARE STATEMENT.						
g		000103	automatic		float bin(15)	initial dcl 2 set ref 2 7 2
range		000102	automatic		float bin(15)	dcl 2 set ref 7 8
theta		000101	automatic		float bin(15)	dcl 2 set ref 5 7 8
v0		000100	automatic		float bin(15)	dcl 2 set ref 5 6 7 8
NAME DECLARED BY EXPLICIT CONTEXT.						
RANGE		000037	constant		entry	external dcl 1 ref 1
NAMES DECLARED BY CONTEXT OR IMPLICATION.						
sind					builtin function	internal ref 7
sysin		000016	constant		file	set ref 0 5
sysprint		000014	constant		file	set ref 0 8 9

STORAGE REQUIREMENTS FOR THIS PROGRAM.

	Object	Text	Link	Symbol	Defs	Static
Start	0	0	260	304	171	270
Length	530	171	24	210	66	2

External procedure RANGE uses 150 words of automatic storage

THE FOLLOWING EXTERNAL OPERATORS ARE USED BY THIS PROGRAM.
return ext_entry get_end put_end stream_io put_data_els

NO EXTERNAL ENTRIES ARE CALLED BY THIS PROGRAM.

THE FOLLOWING EXTERNAL VARIABLES ARE USED BY THIS PROGRAM.
sysin sysin.fsb sysprint sysprint.fbs

Offsets and locations of variables and the starting positions and lengths under "STORAGE REQUIREMENTS" are given in octal. Thus the variable 'vo' is stored at decimal location 64 of the procedure's stack frame.

Error Listing

The error listing gives the error diagnostics. The error report of the example 'RANGE' is as follows:

WARNING 75
The undeclared identifier "sysprint" has been contextually declared as a file constant. It will acquire default attributes.

WARNING 75
The undeclared identifier "sysin" has been contextually declared as a file constant. It will acquire default attributes.

Map Listing

The map listing gives the starting locations in the object code for each statement of the source language. The map for the example 'RANGE' is as follows:

LINE	LOC								
1	000036	2	000056	4	000060	5	000104	6	000115
7	000117	8	000131	9	000156	10	000167	11	000170

The locations are given in octal. Thus the decimal location of the statement on line 2 is word 46 of the object segment, word zero being the first word in the object segment.

APPENDIX A

GUIDE TO PL/I STATEMENTS

A brief description of each of the PL/I statements is given in this appendix. The emphasis is on the syntax of the statements rather than on the interpretation. The entries are arranged in alphabetical order according to the initial keyword of each statement.

PRELIMINARY REMARKS

The statement descriptions given in this appendix have three parts: the syntax diagram, the supplementary rules, and the brief interpretation. The syntax diagram is enclosed in a box and uses the special notation described later. In the description of the 'signal' statement that appears in this appendix, the following syntax diagram is given:

```
signal cd ;
```

This diagram defines the 'signal' statement as "the identifier 'signal', followed by a cd, followed by a ';'". The supplementary rules provide information about the statement that is not given in the diagram. For the 'signal' statement, the supplementary rules are just the clause:

where cd must be a condition designator

This clause defines the syntactic variable, cd, so that the diagram can now be interpreted as "the identifier 'signal', followed by a condition designator, followed by a ';'". The brief interpretation discusses the action taken by the statement and refers the reader to the appropriate section(s) of this manual for a complete definition. For the 'signal' statement, the brief interpretation is:

The statement signals the condition designated by cd. See Section XIII, "Condition Handling."

Syntax Notation

The following paragraphs are a complete description of the notation that is used in the syntax diagrams. Many examples are given.

BASIC CONSTRUCTS

In a syntax diagram, an underlined identifier is a syntax variable; it represents a set of constructs that is defined somewhere else, either in another diagram or in the text. Certain other parts of a diagram, described in the following paragraphs, are used for special purposes, such as indicating a list of items. All the remaining characters in the syntax diagram are literal constructs. Thus in the diagram already given for the 'signal' statement, the underlined identifier 'cd' is a syntax variable and 'signal' and ';' are taken literally.

The syntax variables and literal constructs of a diagram are separated from one another by blanks. According to the "Separation Rules" given in Section V, "Program Syntax," these blanks can be replaced by newlines, tabs, and comments; and, in some places, blanks can be inserted or omitted. Thus, for example, a 'signal' statement can be written as:

```
signal
fixedoverflow;
```

This statement was obtained by replacing the first blank as a newline and omitting the blank between the condition designator and the semicolon.

LIST OF ITEMS

In a syntax diagram, the characters ', ...' are not interpreted literally; instead, they indicate a list of items separated from one another by commas. Similarly, the characters '...' indicate a list of items separated by blanks. The items in the list are specified by the construct that precedes the ', ...' or '...'.

As an example, consider the syntax diagram for the assignment statement, which is:

```
target, ... = e ;
```

This rule is a short way of specifying one of the following forms:

```
target = e ;
target , target = e ;
target , target , target , = e ;
(etc.)
```

This notation does not imply that the targets must be identical; for example, a valid assignment statement is:

```
S3(2),X,alpha = M;
```

This statement has three different targets.

As a second example, consider the syntax diagram for the 'free' statement, which is:

```
free { ref1 in(ref2) } , ... ;
```

Here, each item in the list has the form:

```
ref1 in(ref2)
```

This example shows that the construct that precedes the ', ...' can be a sequence of constructs enclosed in curly braces.

CHOICE OF ITEMS

In a syntax diagram, a vertical list of items enclosed in curly braces indicates a choice among those items. As an example, consider the syntax diagram for the 'goto' statement:

```
{ goto } ref ;  
{ go to }
```

This rule is a short way of specifying the following forms:

```
goto ref ;  
go to ref ;
```

OPTIONAL ITEMS

In a syntax diagram, a construct enclosed in square brackets is optional. As an example, consider the syntax diagram for the 'if' statement:

```
if e then ex1 [else ex2 ]
```

This rule is a short way of specifying the following forms:

```
if e then ex1 else ex2  
if e then ex1
```

Not all optional features of statements are marked as such in the syntax diagrams; too many brackets obscure the syntax diagram. However, a construct that is optional is always mentioned in the supplementary rules that follow a syntax diagram. This treatment is accorded primarily to the PL/I constructs called options.

RECURSIVE DIAGRAMS

Occasionally, a diagram that defines a certain syntax variable also contains that syntax variable. As an example, consider the following definition for declaration:

$$[\text{level}] \left\{ \begin{array}{l} \text{id} \\ (\text{declaration}, \dots) \end{array} \right\} [\text{attribute} \dots]$$

In this diagram, the choice of the first alternative in the braces avoids recursion and produces something like:

alpha fixed external

However, the choice of the second alternative introduces a parenthesized list of declarations; for example:

(sysin,sysout) file

or even:

(x, (y,z) float, beta) controlled

Parts of a Statement

A statement is composed of a prefix, followed by statement body, followed by a semicolon. A prefix is composed of a sequence of any number (possibly zero) of condition prefixes followed by a sequence of any number (possibly zero) of label prefixes. For some statements ('declare' and 'default'), a condition prefix must not be used. For some statements ('procedure', 'entry', and 'format') a label prefix must be used.

A condition prefix has the form:

(id, ...)

where each id is an identifier indicating the enabling or disabling of a condition. A label prefix has the form:

id [(int)] :

where id is an identifier and int is an optionally-signed decimal integer.

The parts of a statement are described fully in Section V, "Program Syntax." The role of the condition prefix is defined in Section XIII, "Condition Handling." The role of the label prefix is defined in Section XI, "Program Flow."

Specific Conventions

Every reference or expression mentioned in the statement descriptions must yield a scalar value unless an exception is explicitly noted.

The following syntactic variables are used throughout this appendix with the given meaning:

e, e1, e2, and so on, represents an arbitrary expression

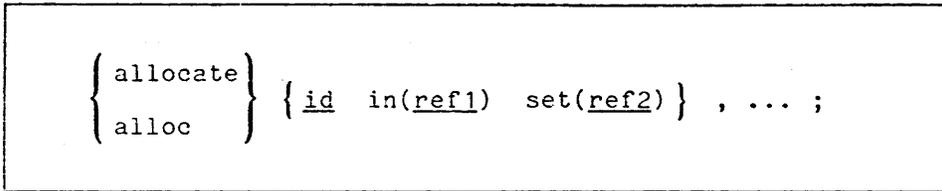
ref, ref1, ref2, and so on, represents an arbitrary reference to a constant, variable, or function.

id represents an arbitrary identifier

target represents an arbitrary variable reference or pseudo-variable

Other syntactic variables are used for specific statements, and their meanings are given in the supplementary rules that follow the diagram in which they are used.

ALLOCATE



where id must be a 'controlled' or 'based' variable name, ref1 must yield an area variable, and ref2 must yield a locator variable. The options are selected as follows:

- If id is 'controlled', then both options must be omitted.
- Suppose id is 'based'. If id is declared 'based(q)', then the 'set' option can be omitted and 'set(q)' is assumed; otherwise, the 'set' option must be given.
- If id is a 'based' variable intended for allocation in system storage rather than in an area variable, then the 'in' option must be omitted.
- Suppose id is a 'based' variable intended for allocation in an area. If ref2 is declared 'offset(a)', then the 'in' option can be omitted and 'in(a)' is assumed; otherwise, the 'in' option must be given.

The options can be given in any order, but the order shown above is recommended.

The statement allocates storage. If id is 'controlled', storage is allocated in system storage and is stacked on any previous allocations of id. If id is 'based', storage is allocated either in system storage or the area given by the 'in' option; then the locator variable given in the 'set' option is set to designate the allocated storage.

ASSIGN

```
target, ... = e ;
```

The expression e is evaluated and its value is assigned to the storage specified by each target. The expression e can be any kind of PL/I expression, scalar or aggregate. A target can be a variable reference or a pseudo-variable. Each target must designate a storage unit that can accommodate the value of e after the value has been subjected to an allowed conversion of its storage type. See Section X, "Value Assignment."

BEGIN

```
begin [options (non_quick)] ;
```

The statement is the first statement of a 'begin' block. A 'begin' block has two purposes: it groups together the statements contained in the block and it delimits the scope of the names declared in the block. The 'options(non_quick)' attribute may be used with the 'begin' statement. See Sections V and XI, "Program Syntax" and "Program Flow," respectively.

CALL

```
call ref( e, ... ) ;
```

where, if there are no arguments, the parenthesized argument list, '(e, ...)', can be omitted or written as '()'.

The statement invokes a procedure. The ref is evaluated to give an entry value. A procedure block is entered at the 'procedure' statement or 'entry' statement designated by the value of ref and the procedure is executed for the given argument list. The argument list must contain one argument for each parameter in the statement at which the procedure is entered. The procedure must not return a value for an invocation by a 'call' statement. See Section XII, "Procedure Invocation."

CLOSE

```
close { file(ref) } , ... ;
```

The statement closes one or more files. Each ref specifies a file value, and the corresponding file is closed if it is not already closed. See Sections XIV and XV, "Stream Input/Output" and "Record Input/Output," respectively.

DECLARE

```
{ declare  
  dcl } declaration, ... ;
```

where:

declaration is defined as

```
[ level ] { id  
             ( declaration, ... ) } [ attribute ... ]
```

and where level is an unsigned decimal integer. The statement must not be preceded by a condition prefix.

The statement declares one or more identifiers. An identifier is declared by associating with it a set of attributes and, in the case of a component of a structure, a level number. See Section VI, "Declarations."

DEFAULT

The 'default' statement allows a programmer to extend and modify the defaults that are built into PL/I for the declaration of identifiers; in addition, it allows the programmer to specify declarations that he wishes to have flagged as errors even though they are otherwise valid in PL/I. The 'default' statement is not recommended for use in an individual program; rather, its application lies in the establishment and maintenance of special standards for all programs written for a given project or at a given computing installation. The 'default' statement is not described here, but a full description appears in the PL/I Language Specification.

DELETE

```
delete file(ref) key(e) ;
```

The options are selected as follows:

- o The 'key' option must be omitted if the file is not 'keyed'.
- o The 'key' option cannot be omitted if the file is 'direct'.

The options can be given in any order, but the order shown above is recommended.

The statement deletes a record from a data set. The record is specified by the file value given by the 'file' option and the character-string value given by the 'key' option (if present). See Section XV, "Record Input/Output."

$$\left\{ \begin{array}{l} \text{do ;} \\ \text{do while(e1) ;} \\ \text{multiple-do ;} \end{array} \right\}$$

where:

multiple-do is defined as

$$\text{do target} = \left\{ \begin{array}{l} \underline{e2} \text{ while}(\underline{e3}) \\ \underline{e4} \text{ repeat } \underline{e5} \text{ while}(\underline{e3}) \\ \underline{e6} \text{ by } \underline{e7} \text{ to } \underline{e8} \text{ while}(\underline{e3}) \end{array} \right\}, \dots$$

The options and clauses are selected as follows:

- The option 'while(e3)' can be omitted. In that case, 'while("1"b)' is assumed.
- The 'by e7' clause can be omitted. In this case, 'by 1' is assumed.
- The 'to e8' clause can be omitted, provided that the 'by' clause is not omitted. In this case, the end test associated with the 'to' clause is not performed.

The 'by' clause and the 'to' clause can be interchanged.

The first of the three forms, 'do;', is the first statement of a noniterative group. The statements contained in the group are executed exactly once.

The second of the three forms, 'do while(e1):', is the first statement of an unindexed iterative group. Execution of the group begins with a while test. The expression e1 is evaluated and must yield a bit-string value. If all of the bits are zero bits, then execution is complete; otherwise, the statements in the group are executed and another execution of the group begins (with another while test).

The third of the three forms, 'do target = ...', is the first statement of an indexed iterative group. The statement provides an index, specified by target, that is used to control the repeated execution of the statements in the group. As the second syntax diagram shows, there are three ways to control the index; their interpretation is not given here. See Section XI, "Program Flow."

END

```
end [ id ] ;
```

This statement is the last statement of a group, a 'begin' block, or a procedure. The optional identifier id is a closure label; its use is not recommended. See Sections V, XI, and XII, "Program Syntax", "Program Flow", and "Procedure Invocation," respectively.

ENTRY

```
entry( id, ... ) [ returns( dec1 ) ] ;
```

where:

- The statement must be immediately contained in a procedure.
- The statement must begin with at least one label prefix; however, no label prefix can have a subscript.
- If there are no parameters, the parameter list '(id, ...)' can either be omitted or written as '()'.
- Each identifier in the parameter list must be a level-one variable name declared in the immediately containing procedure.
- The number of parameters in the parameter list must equal the number of arguments in the argument list of the 'call' statement or function reference that invoked this entry.
- The 'returns' attribute must be omitted or must appear depending on whether the entry is invoked by a 'call' statement or a function reference.
- dec1 is the declaration of the value that is returned by the procedure when it is invoked at this entry.

The statement provides an additional entry to a procedure. The entry can differ from other entries in its position within the procedure, in its parameter list, and in its 'return' attribute. Thus, one procedure can be invoked in several different ways if 'entry' statements appear in the procedure.

FORMAT

```
format ( format-list ) ;
```

where the statement must be preceded by at least one label-prefix and:

```
format-list is defined as  
  
{ format-item  
  [ n ] { format-item }  
  ( e ) ( format-list ) } , ...
```

where n is an unsigned integer and the format-items are as follows:

- a(w) -- transmit a character string value representation
- b(w) -- transmit a bit string value representation
- f(w,fw,dm) -- transmit a fixed-point value representation
- e(w,fw,ms) -- transmit a floating-point value representation
- c(part1,part2) -- transmit a complex value representation
- p"x" -- transmit a pictured value representation
- x(e) -- skip e character positions
- column(e) -- skip to column e of a line
- skip(e) -- skip e lines
- line(e) -- skip to line e of a page
- page -- skip to the next page

In these format items, w (width) is the number of characters in an input or output field, fw (fraction width) is the number of fractional digits in the value representation, dm (decimal multiplier) is a power of ten, ms (mantissa significance) is the number of digits in the mantissa of a floating-point value representation, part1 and part2 can each be any 'f', 'e', or 'p' format item, x must be a PL/I picture, e is an arithmetic expression, and ref is a format-valued reference. Some of the arguments of format-items can be omitted and default values are then assumed.

The statement supplies a format-list for use in an edit-directed stream input/output statement. The format-list provides a format item for each value transmitted and also provides format items to skip spaces, lines, and pages between value representations. See Section XIV, "Stream Input/Output."

FREE

```
free  {ref1 in(ref2)}  , ... ;
```

The option is selected as follows:

- The 'in' option must be omitted if the storage being freed is a controlled variable.
- The 'in' option must be omitted if the storage being freed is a based variable that is allocated in system storage.
- The 'in' option can be omitted in any case in Multics PL/I; but Standard PL/I requires the option for a based variable allocated in an area.

The statement frees the controlled or based variable designated by ref1. See Section VII, "Storage Management."

GET

```
get { file(ref1) copy(ref2) skip(e1) } input-option ;  
    { string(e2) copy(ref2) }
```

where:

input-option is defined as

```
{ data( data-ref, ... )  
  list( get-item, ... )  
  edit { ( get-item, ... ) ( format-list ) } ... }
```

get-item is defined as

```
{ target  
  ( get-item, ... multiple-do ) }
```

and where:

- The get-item for a 'data' option is restricted to a simple or unsubscripted structure-qualified variable reference.
- The format-list is defined under the 'format' statement in this appendix.
- The multiple-do is defined under the 'do' statement in this appendix.

The options are selected as follows:

- The 'file' and 'string' options can be omitted. In this case, 'file(sysin)' is assumed.
- The 'copy' option can be omitted. In this case, no copy of the input is made.
- The parenthesized argument '(ref2)' in the 'copy' option can be omitted. In this case, 'copy(sysprint)' is assumed.
- Either the 'skip' option or the input option (but not both) can be omitted. In this case, the corresponding skip or input is not performed.

- The parenthesized argument '(e1)' in the 'skip' option can be omitted. In this case, 'skip(1)' is assumed.
- The list of input items '(data-ref, ...)' can be omitted from the 'data' input option. In this case, a list of input items including every level-one variable accessible from the 'get' statement is assumed. This variation is expensive, and should not be used without due consideration of the cost.

The options can be given in any order, but the order shown above is recommended.

The statement reads value representations from a stream data set and assigns their values to program variables. See Section XIV, "Stream Input/Output."

GOTO

$\left. \begin{array}{l} \text{goto} \\ \text{go to} \end{array} \right\} \text{ ref ;$

The statement transfers control to the statement specified by the value of ref. See Section XI, "Program Flow."

IF

```
if e then ex1 [else ex2]
```

where ex1 and ex2 are executable units. An executable unit is defined as:

a group (a 'do' statement, other statements, and an 'end')
a begin-block (a 'begin' statement, other statements, and an 'end')
an independent statement (any statement which acts alone).

An independent statement is any statement except the following:

declarative statements: 'declare' and 'default'
dependent statements: 'format', 'do', 'begin', 'procedure', 'entry', 'end'.

The 'else ex2' clause can be omitted, giving a statement of the form:

```
if e then ex1
```

The statement evaluates e to produce a bit-string value. If the bit-string value contains a '1' anywhere (and therefore represents "true"), then ex1 is executed and ex2 is skipped. If the bit-string contains only '0' bits, then ex1 is skipped and ex2 is executed. If the 'else' clause is omitted, then no action is performed when the bit-string contains only '0' bits.

LOCATE

```
locate id set(ref1) file(ref2) keyfrom(e) ;
```

The options can be omitted as follows:

- The 'set' option can be omitted provided that id is declared (elsewhere) as 'based(q)', where q is a locator qualifier. In this case, 'set(q)' is assumed.
- The 'file' option cannot be omitted.
- The 'keyfrom' option must be omitted if the file designated by ref1 does not have the 'keyed' attribute.

The options can be given in any order, but the order shown above is recommended.

The statement allocates storage for the based variable id and sets the locator variable specified by ref1. The allocated storage serves as a buffer, and values can be assigned to it by subsequent statements. Then, when the next output statement is encountered (or the file is closed), the contents of the buffer are output. Output is directed to the file designated by ref2 using the key given by e (if the 'keyfrom' option occurs). See Section XV, "Record Input/Output."

NULL

```
;
```

The statement does nothing. It can be used where an executable-unit is called for, as in the 'if' and 'on' statements. See Sections XI and XIII, "Program Flow" and "Condition Handling", respectively.

ON

```
on cr-list [snap] 'on' unit ;
```

where cr-list is a list of condition references separated by commas and 'on' unit is:

```
a begin block  
a statement  
the keyword 'system'
```

If the 'on' unit is a 'begin' block, then a 'return' statement can appear only if it is within a procedure within the 'begin' block. If the 'on' unit is a statement, then it must not be any of the following:

```
declarative statements: 'declare' and 'default'.  
dependent statements: 'format', 'do', 'begin', 'procedure', 'entry', 'end'.  
other excluded statements: 'on', 'revert', 'if', 'return'.
```

If an 'on' statement begins with a condition prefix, that prefix applies only to the evaluation of the condition name. If the 'on' unit is a 'begin' block or a statement, it can have its own condition prefixes, but it cannot have a label prefix.

The statement establishes the 'on' unit as the action which will be taken when any conditions in cr-list are signalled. If 'snap' appears, the Multics probe command is called if the program is running interactively. If the program is running in absentee, the trace stack command is called. Therefore, either probe or trace stack is called just before the 'on' unit is invoked. If 'system' is used as the 'on' unit, the PL/I default 'on' unit for the condition is invoked. See Section XIII, "Condition Handling."

OPEN

```
open {file(ref1) title(e1) linesize(e2) pagesize(e3) fd} , ... ;
```

where:

where fd (the file description) is defined as:

```
stream { input
        { output [print] [environment (interactive)] } }

record { sequential } { input
        { keyed sequential } { output } environment(stringvalue)
        { keyed direct } { update }
```

The options and attributes are selected as follows:

- The 'file' option cannot be omitted.
- The 'title' option can be omitted. In this case, 'title(vfile fn)' is assumed, where fn is the file name specified by the 'file' option.
- The 'linesize' and 'pagesize' options must be omitted unless the file description includes 'output print'.
- The 'linesize' option can be omitted when the file description includes 'output print'. In this case, 'linesize(132)' is assumed for most devices (including the printer) but the line size is device-dependent for a terminal that uses the tty_ I/O module.
- The 'pagesize' option can be omitted when the file description includes 'output print'. In this case, 'pagesize(60)' is assumed for most devices (including the printer) but the page size is infinity for a terminal that uses the tty_ I/O module.
- The 'stream' and 'record' attributes can be omitted because they can be deduced from other attributes given in the file description.
- The 'output' attribute can be omitted if the file description contains the 'print' attribute.
- The 'environment' attribute can be abbreviated to 'env'.
- The 'keyed' attribute can be omitted if the file description includes the 'direct' attribute.

The statement supplies information to control the transmission of data between a data set and program storage. The information is stored in a file-state block designated by the file value supplied by ref1 in the 'file' option. The data set to be used is designated by the character-string value supplied by e1 in the 'title' option. The format of a printed page of output is specified by integers supplied by e2 and e3. The intended use of the data set is specified by the file description. See Sections XIV and XV, on "Stream Input/Output" and "Record Input/Output," respectively.

PROCEDURE

```
procedure
proc      ( id, ... ) [attribute ...][recursive] ;
```

where:

- The statement is the first statement of a procedure block.
- The statement must begin with at least one label prefix; however, no label prefix can have a subscript.
- If there are no parameters, the parameters list '(id, ...)' can either be omitted or written as '()'.
- Each identifier in the parameter list must be a level-one variable declared immediately within the procedure.
- The number of parameters in the parameter list must equal the number of arguments in the argument list of the 'call' statement or function reference that invoked this entry.
- The 'returns' attribute must be omitted or must appear according to how the procedure is invoked by a 'call' statement or a function reference. If the 'reducible' attribute is given, 'irreducible' cannot be given.

The 'option' attribute may not specify the keyword 'constant'. It may specify 'support' only if the 'procedure' statement heads an 'external procedure'. It also may specify the word 'non_quick'. If the 'options(main)' attribute appears in an 'external procedure' statement and it is the first procedure invoked in a run unit, then the 'external procedure' is considered the 'main procedure'.

- decl is the declaration of the value which is returned by the procedure when it is invoked at this entry.
- The 'recursive' keyword must be used in Standard PL/I if the procedure is recursive. This keyword is ignored by Multics PL/I, which assumes that all procedures are recursive.

The order in which the attribute and 'recursive' keyword are given above is recommended, but the reverse order is permitted.

The statement provides the principal entry to a procedure. One or more 'entry' statements can be used to provide additional entries to the same procedure. See Section XII, "Procedure Invocation."

PUT

$$\text{put } \left\{ \begin{array}{l} \text{file}(\underline{\text{ref1}}) \\ \text{string}(\underline{\text{target}}) \end{array} \right\} \left\{ \begin{array}{l} \text{skip}(\underline{\text{e1}}) \\ \text{page } \text{line}(\underline{\text{e2}}) \end{array} \right\} \underline{\text{output-option}} ;$$

where:

output-option is defined as

$$\left\{ \begin{array}{l} \text{data}(\underline{\text{put-item}}, \dots) \\ \text{list}(\underline{\text{put-item}}, \dots) \\ \text{edit } \left\{ (\underline{\text{put-item}}, \dots) (\underline{\text{format-list}}) \right\} \dots \end{array} \right\}$$

put-item is defined as

$$\left\{ \begin{array}{l} \underline{\text{e3}} \\ (\underline{\text{put-item}}, \dots \underline{\text{multiple-do}}) \end{array} \right\}$$

and where:

- The expression, e3, in a put-item for a 'data' option is restricted to a variable reference.
- The format-list is defined under the 'format' statement in this appendix.
- The multiple-do is defined under the 'do' statement in this appendix.

The options are selected as follows:

- The 'file' and 'string' options can be omitted. In this case, 'file (sysprint)' is assumed.
- The 'skip' option, 'page' option, 'line' option, and output-option can be omitted in any way provided one of them remains. In these cases, the corresponding output operation will not occur.
- The parenthesized argument '(e1)' in the 'skip' option can be omitted. In this case, 'skip(1)' is assumed.
- The list of output items '(put-item, ...)' can be omitted from the 'data' output option. In this case, a list of output items including every level on variable accessible from the 'put' statement is assumed.

The options can be given in any order, but the order shown in the syntax diagram is recommended.

The statement takes values from program storage (or computes values) and writes their representations into a stream data set. See Section XIV, "Stream Input/Output."

READ

$\text{read file}(\underline{\text{ref1}}) \left\{ \begin{array}{l} \text{key}(\underline{\text{e1}}) \\ \text{keyto}(\underline{\text{target}}) \end{array} \right\} \left\{ \begin{array}{l} \text{into}(\underline{\text{ref2}}) \\ \text{set}(\underline{\text{ref3}}) \\ \text{ignore}(\underline{\text{e2}}) \end{array} \right\} ;$
--

The options are selected as follows:

- The 'key' option must be omitted if the file is not 'keyed' or if the 'ignore' option is used.
- The 'key' option cannot be omitted if the file is 'direct'.
- The 'keyto' option must be omitted if the file is not 'keyed sequential' or if the 'ignore' option is used.

The options can be given in any order, but the order shown above is recommended.

The statement reads a record from a data set or skips records in a sequential data set. The data set is specified by the file value given in the 'file' option. The 'key' option gives a character-string value used to locate a record. The 'keyto' option gives a storage unit suitable for a character-string value representing a key. The 'into' option provides storage for an input value. The 'set' option provides storage for a pointer to a system buffer into which (if this option is used) the record will be read. The 'ignore' option gives the number of records to be skipped as an integer value. See Section XV, "Record Input/Output."

RETURN

```
return [ ( e ) ] ;
```

The parenthesized expression must appear if the statement returns to a function reference, and must be omitted if the statement returns to a procedure call.

The statement terminates execution of a procedure and returns control to the function reference or 'call' statement that invoked it. If this is a return from a 'main procedure', then this is the same as a 'stop' statement. See Section XII, "Procedure Invocation."

REVERT

```
revert cr, ... ;
```

where cr must be a condition reference.

The statement reverts each condition name provided it was established by an 'on' statement in the current block. See Section XIII, "Condition Handling."

REWRITE

```
rewrite file(ref1) key(e1) from(ref2) ;
```

The options are selected as follows:

- The 'key' option must be omitted if the file is not 'keyed'.
- The 'key' option cannot be omitted if the file is 'direct'.

The options can be given in any order, but the order shown above is recommended.

The statement replaces a record in a data set. The record to be replaced is specified by the file value given by the 'file' option and the character-string value given by the 'key' option. The new value for the record is supplied by the reference in the 'from' option. See Section XV, "Record Input/Output."

SIGNAL

```
signal cr ;
```

where cr must be a condition reference.

The statement signals the condition designated by cr. See Section XIII, "Condition Handling."

STOP

```
stop ;
```

This statement terminates the program if executed within a run unit. If it is not executed within a run unit, control is returned to the Multics command processor in such a way that the remainder of the command line is executed.

WRITE

```
write file(ref1) keyfrom(e) from(ref2) ;
```

The options are selected as follows:

- The 'key' option must be omitted if the file is not 'keyed'.
- The 'key' option cannot be omitted if the file is 'direct'.

The options can be given in any order, but the order shown above is recommended.

The statement adds a record to a data set. The destination of the record is specified by the file value given by the 'file' option and the character-string value given by the 'keyfrom' option (if present). The value for the record is supplied by the reference in the 'from' option. See Section XV, "Record Input/Output."

APPENDIX B

NEW FEATURES

This appendix has been added to the manual to describe two new features that will be installed for MR7.0. They are: unsigned binary and 4-bit decimal. For the next revision of the manual, these features will be incorporated into text and all related topics will reflect their addition. At the present, however, the text of the main part of the manual has not been changed.

'signed' AND 'unsigned' ATTRIBUTES

The 'signed' and 'unsigned' attributes are both nonstandard and both only apply to arithmetic values. They influence the representation of values in storage. Real arithmetic data may be stored in variables with or without a sign and storage sharing requires that the variables sharing storage are either 'signed' or 'unsigned', as well as data types and alignment.

The 'signed' attribute has the following constraints:

- 'signed' arithmetic variables always contain storage to represent the sign of their value
- 'signed' is assumed if 'unsigned' is not specified
- 'signed' may be specified for 'fixed' or 'float' variables

The 'unsigned' attribute has the following keyword and abbreviation:

<u>Keyword</u>	<u>Abbreviation</u>
unsigned	uns

and has the following constraints:

- 'unsigned', unaligned arithmetic variables do not contain storage to represent the sign of their value
- 'unsigned' must be specified; it is not assumed
- 'unsigned' variables may only contain nonnegative values
- 'unsigned' variables may only be specified for 'real', 'fixed', 'binary' variables
- the 'size' condition occurs when a negative value is assigned to a variable whose declaration contains the 'unsigned' attribute

'9-bit' AND '4-bit' DECIMAL

In Multics PL/I, decimal data may be stored in one of two ways: as '9-bit' decimal or as '4-bit' decimal. Aligned decimal data, which is the default, is stored as '9-bit' decimal with each digit occupying nine bits. Unaligned decimal data is stored as '4-bit' decimal with each pair of digits occupying nine bits. For a more complete description of standard data types, see "Standard Data Type Formats" in Appendix D of the MPM Reference Guide. In that discussion, descriptor types 9 through 12 apply to PL/I '9-bit' decimal and descriptor types 44 through 46 apply to PL/I '4-bit' decimal. Also, for a more detailed discussion of data descriptors, see "Subroutine Calling Sequences" in Section II of the MPM Subsystem Writers' Guide.

INDEX

MISCELLANEOUS

- \$ indicator
 - drifting 3-37
 - leftmost 3-34
- %include macros 5-7
- & operator 9-41
- * operator 9-9
- + indicator
 - drifting 3-36
 - leftmost 3-33
- + operator
 - infix 9-6
- + operator prefix 9-6
- , indicator 3-38
- indicator
 - drifting 3-36
 - leftmost 3-33
- operator
 - infix 9-6
- operator prefix 9-6
- brief option 16-15
- brief_table option 16-15
- check option 16-15
- extend option 16-11
- length option 16-12
- list option 16-15
- map option 16-15
- nnl option 16-12
- optimize option 16-15
- profile option 16-15, 16-18
- source option 16-15
- symbols option 16-15
- table option 16-15, 16-17
- . indicator 3-38
- / in comment 5-10
- / indicator 3-38
- / operator 9-8
- // operator 9-32
- 9 indicator
 - nonnumeric 3-43
 - numeric 3-30
- < operator
 - for arithmetic values 9-10
 - for string values 9-37
- <= operator
 - for arithmetic values 9-10
 - for string values 9-37
- = operator
 - for address values 9-48
 - for arithmetic values 9-10
 - for string values 9-37
- > operator
 - for arithmetic values 9-10
 - for string values 9-37
- >= operator
 - for arithmetic values 9-10
 - for string values 9-37

A

- a format item 14-29
- a indicator 3-43
- abbreviations and defaults
 - alignment attributes 3-63
 - area attribute 3-48
 - arithmetic attributes 3-8
 - definition of default 3-9
 - dimension attribute 3-55

- abbreviations and defaults (cont)
 - guidelines for use 3-8
 - line length 14-11
 - management class attributes 7-11
 - ordinary string attributes 3-17
 - page length 14-11
 - picture attributes 3-23
 - shortened variable references 8-25
- abbreviations and defaults files 14-12
- abs function 9-16
- absolute pathname 16-1
- activation indexes 12-35
- activation internal regions 7-4
- activation pointer 12-36
- activation regions
 - for procedure activation 12-10
 - in recursion 12-28
- activation variable reference 12-38
- add function 9-10
- addr function 7-25, 9-49
- addrel function 9-52
- addresses
 - assignment of 10-2
 - attributes 3-45
 - implicit locator targets 4-6
 - operations 9-48
 - values 2-3
- after function 9-35
- aggregate
 - assignment of 10-3
 - expressions 8-5
- aggregates
 - aggregate type 3-49
 - role of 3-2
 - guidelines for use 3-60
 - operations on 9-2
 - values 2-4
- algebraic comparison operators 9-10
- aligned attribute 3-62
- alignment types
 - attributes 3-61
 - role of 3-2
- allocate statement
 - for based variables 7-22
 - for controlled variables 7-18
 - syntax diagram A-5
- allocation function 9-72
- allocation of storage 7-1
- ambiguous declaration 6-19
- and operator 9-41
- applicability of declarations 6-17
- approximation of arithmetic values for conversion 4-8
- area
 - assignment of 10-3, 10-7
 - attributes 3-47
 - condition 7-40
 - initialization of 7-6
 - operations 9-48
 - size 3-47
 - storage 3-46
 - values 2-4
- area function 9-53
- argument evaluation in operations 9-2
- arguments 12-2
 - by-reference 12-3
 - by-value 12-3
 - classification of 12-3
 - connected and unconnected 12-5
 - guidelines for 12-8
 - interpretation of 12-6, 12-10
 - passing of 12-2
- arithmetic
 - assignment 10-1
 - constant literals 8-29
 - constants
 - as lexemes 5-5
 - data types 3-5
 - guidelines for data 3-14
 - operations 9-3
 - storage 3-4
 - values 2-1
- arithmetic decimal-point indicator 3-39
- array
 - values 2-4
- arrays 3-53
 - dimension attribute 3-54
 - operations 9-53
 - storage 3-57
 - storage layout 3-71
 - storage types 3-56
- ASCII characters 2-2, 3-16
 - new line 14-2
 - new page 14-2
- assignment statements 10-1
 - area assignments 10-7
 - form of 10-4
 - guidelines for 10-8
 - interpretation of 10-5

- assignment statements (cont)
 - order of interpretation 10-7
 - overlapping string targets 10-5
 - pseudo-variables 10-9
 - restrictions 10-5
 - special string target 10-6
 - syntax diagram A-6
- associated storage types 8-21
- asterisk extent 12-7
- atan function
 - Cartesian 9-26
 - ordinary 9-25
- atand function
 - Cartesian 9-27
 - ordinary 9-26
- atanh function 9-29
- attach description 16-10
 - for record files 15-5
 - for stream files 14-7
- attaching files 16-9
- attributes 6-20
 - classification of 6-24
 - complete sets 6-20
 - for built-in function names 6-23
 - for condition names 6-23
 - for constant literals 8-34
 - for constant names 6-22, 8-39
 - for generic names 6-23
 - for variable names 6-21
- automatic variables 7-16
 - attribute 7-10
 - guidelines for 7-32

B

- b format item 14-29
- b indicator 3-38
- base attribute
 - guidelines for choice of 3-15
- base attributes 3-6
- base variable 7-25
- based input 15-16
- based output 15-20
- based variables 7-20
 - attribute 7-10
 - guidelines for 7-34
- baseno function 9-52
- baseptr function 9-51
- before function 9-34
- begin block
 - as executable unit 11-4, 11-19
 - as program structure 5-18
 - in general recursion 12-40
- begin statement
 - syntax diagram A-6
- binary
 - attribute 3-6
 - function 9-62
- bind command 7-14, 16-7
- binding 16-7
- bit 3-16
 - attribute 3-16
 - function 9-65
 - operations 9-41
- bit values 2-3
- block
 - activation of 11-19
 - as program structure 5-18
 - exit from 11-16
- BNF syntax 5-1
- body
 - of do group 11-9
- body of statement 5-14
- books on PL/I 1-12
- bool function 9-42
- Boolean values 2-3
- bounds of array 3-54
- box of storage unit 3-2
- break character 5-3
- built-in function names 8-43
 - attributes for 6-23
- built-in function references 8-42
- by-reference arguments 12-3
- by-value arguments 12-3

C

- c format item 14-34
- call statement 12-9
 - execution of 12-9
 - syntax diagram A-6

- capacity of storage 7-37
- ceil function 9-12
- chained recursion 12-28
- character
 - attribute 3-16
 - function 9-64
 - operations 9-45
 - values 2-2
- characters 2-2, 3-16
 - classification of 5-2
 - escape conventions 8-34
- classification
 - of arguments 12-3
 - of attributes 6-24
 - of characters 5-2
 - of conditions 13-2
 - of lexemes 5-12
 - of operators 5-6
 - of picture indicators 3-21
 - of pictures 3-22
 - of statements 5-16
 - of values 2-5
- clause
 - declaration 6-5
- clauses
 - general form 5-14
- close statement
 - for record files 15-10
 - for streams 14-11
- close statements
 - syntax diagram A-7
- collate function 9-45
- collating sequence 9-39, 9-46
- column format item 14-40
- column position 14-5
 - initialization of 14-10
- combined declarations 6-6
- command level
 - attachment 16-9
- comments
 - as lexemes 5-10
- common data attributes 9-5
- comparison operations 9-37
 - for value addresses 9-48
- comparison operators
 - for arithmetic values 9-10
- compiler 16-3
- compiling an external procedure 16-14
- complete attribute sets 6-20
- complex
 - attribute 3-5, 3-23
 - function 9-18, 9-60
- complex arithmetic function 9-17
- complex format items 14-34
- complexvalues 2-1
- component names 16-2
- components of aggregate 3-49
- compound list items 14-22
 - array variable names 14-22
 - iterated list 14-23
 - structure variable names 14-23
- computational conditions 13-2
- computational values 2-5
- concatenate operator 9-32
- condition
 - attribute 13-6
- condition names
 - attributes for 6-23
- condition prefix 13-7
 - purpose of 5-13
 - scope of 13-8
- conditions 13-1
 - built-in functions 13-15
 - classification of 13-2
 - declarations of 13-7
 - default enabling 13-9
 - enabling and disabling 13-7
 - fatal 13-13
 - for conversion 4-20
 - for storage management 7-39
 - for stream input/output 14-44
 - guidelines for 13-17
 - language-defined 13-2
 - computational 13-2
 - input/output 13-4
 - storage 13-3
 - termination 13-3
 - occurrence of 13-12
 - principal features 13-1
 - programmer-defined 13-5
 - references 13-6
 - signalling of 13-12
- conjg function 9-19
- consequences of if statement 11-4
- constant literals 8-29
 - attributes for 8-34
 - for arithmetic values 8-29

- constant literals (cont)
 - for string values 8-32
- constant names
 - attributes for 6-22
- constant reference
 - as destination of goto 11-16
- constant references 8-35
 - to external entries 8-37
 - to files 8-38
 - to record files 15-4
 - to statement addresses 8-35
 - to stream files 14-6
- constant storage 7-13
 - attribute 7-10
- containing references 8-15
- containment
 - definition of 6-3
- contents
 - of storage unit 3-2
- contextual declaration 6-10
- control block pointer 16-10
- control characters for streams 14-2
- control format items 14-39
- control list in do statement 11-10
- controlled variables 7-18
 - attribute 7-10
 - guidelines for 7-33
 - stacking of 7-18
- conversion condition 4-21, 14-45
 - for arithmetic target 4-7
 - for bit-string target 4-16
- conversion of values 4-1
 - guidelines for functions 9-69
 - conditions for 4-20
 - contexts 4-1
 - arguments and results 4-3
 - assignment statements 4-2
 - assignment-like contexts 4-3
 - bit-string targets 4-6
 - built-ins and expressions 4-4
 - character-string targets 4-5
 - integer targets 4-5
 - locator targets 4-6
 - operations for 9-56
 - targets 4-2
 - to aggregate targets 4-18
 - to arithmetic targets 4-7, 9-59
 - to bit-string targets 4-16, 9-64
 - to character-string targets 4-10, 9-64
 - to locator targets 4-18, 9-66
- convert function 9-57
- copy function 9-32
- copy option
 - in get statement 14-13
- cos function 9-24
- cosd function 9-25
- cosh function 9-28
- cr indicator 3-33
- credit indicator 3-33
- cross-section of array 8-11
- current record indicator 15-2, 15-4
 - initialization of 15-8

D

- dangling else clause 11-7
- data
 - description of 1-2, 3-1
 - storage 3-1
 - values 2-1
- data address values 2-5
- data format items 14-28
- data frame
 - of storage unit 3-10
- data set designator 14-5, 15-4
- data sets
 - record 15-2
 - stream 14-1
- data types
 - addresses 3-45
 - areas 3-47
 - arithmetic 3-5
 - complete attribute sets 6-21
 - conversion of 4-1
 - ordinary strings 3-16
 - pictured strings 3-20
 - role of 3-2
- data-directed input/output 14-16
- date function 9-71
- db indicator 3-33
- debit indicator 3-33
- decap function 9-36
- decimal
 - attribute 3-6
 - function 9-63

- decimal-point indicator 3-32
- declaration 6-1
 - abbreviations and defaults 6-6
 - ambiguous 6-19
 - applicability 6-17
 - clause of 6-5
 - combined declarations 6-6
 - contextual 6-10
 - establishment of 6-1
 - factored declaration 6-6
 - implicit 6-10
 - of built-in function names 6-23
 - of condition names 6-23, 14-7
 - of constant names 6-22
 - of entry names 6-8
 - of format names 6-9
 - of generic names 6-23
 - of label names 6-8
 - resolution of names 6-15
 - short forms of 6-6
 - structure declarations 6-5
- declarations
 - of variable names 6-21
 - simple declarations 6-4
- declare statement 6-4
 - guidelines for 6-7
 - sequential execution of 11-2
 - syntax diagram A-7
- default statement 6-13
 - sequential execution of 11-2
- default statements
 - syntax diagram A-8
- defined variables 7-29
 - attribute 7-10
 - guidelines for 7-34
- delete statement
 - for keyed input/output 15-12
 - for sequential input/output 15-14
- delete statements
 - syntax diagram A-8
- descriptors
 - for returned result 12-19
- designator of storage unit 3-2
- destination of goto statement 11-18
- digit indicator 3-22
- digits 5-2
- dimension attribute 3-54
- dimension function 9-54
- dimensionality of array 3-54
- direct attribute 15-9
- direct recursion 12-28
- directory segment 16-1
- disabling conditions 13-7
- divide function 9-10
- division operator 9-8
- do control
 - FORTRAN 11-13
 - index of 11-15
 - repeat 11-12
 - single value 11-11
- do group 11-8
 - as consequence of if 11-6
 - as executable unit 11-4
 - as program structure 5-17
 - body of 11-9
 - index of 11-15
 - iterative with index 11-10
 - iterative without index 11-9
 - noniterative 11-16
- do statement
 - syntax diagram A-9
- dollar indicator 3-34
- dollar sign 5-3
- dot function 9-56
- double precision 3-7
- drifting-dollar indicator 3-37
- drifting-sign indicator 3-36
- dynamic linking 16-5

E

- e (mathematical quantity) 9-21
- e format item 14-32
- e indicator 3-41
- edit-directed input/output 14-26
- editors 16-14
- edm editor 16-14
- elementary arithmetic operations 9-5
- elements of arrays 3-53
- else clause 11-3, 11-7
- empty function 9-53
- enabling conditions 13-7

- enabling conditions (cont)
 - for debugging 13-18
- end statement 12-22
 - as procedure exit 12-11
 - syntax diagram A-10
- end-round repetitions 14-41
- endfile condition
 - for record input/output 15-24
 - for stream input/output 14-45
- endpage condition 14-46
- entering an external procedure 16-14
- entry
 - attribute 3-45
 - constant names 8-35
 - declaration of 8-37
- entry name versus segment name 16-3
- entry names
 - generic 12-27
- entry points
 - in another external procedure 12-23
 - in same external procedure 12-23
 - outside the program 12-24
- entry reference 12-22
 - constant 12-23
 - interpretation of 12-10
- entry references
 - function 12-26
 - variable 12-25
- entry statement 12-21
 - syntax diagram A-10
- environment attribute 14-11, 15-9
- equivalenced based variables 7-25
- erf function 9-30
- erfc function 9-30
- error condition 13-20
- error function 9-30
- escape conventions
 - for characters 8-34
- established on unit 13-13
- establishing on units 13-10
- establishment of declarations 6-1
- executable procedure segment 16-4
 - name 16-3
- executable unit 11-4
- executing a program 16-15
- exit from block 11-16
- exp function 9-21
- exponent
 - of float value 3-6
- exponentiation operator 9-9
- expressions 8-1
 - aggregate 8-5
 - infix 8-47
 - nested 8-2
 - operator expression 8-46
 - parenthesized 8-3
 - prefix 8-47
 - priority of operators 8-48
- extent 12-7
- extent functions 9-53
- extents
 - evaluation of 7-6
 - for areas 3-47
 - for arrays 3-55
 - for automatic variables 7-16
 - for based variables 7-23
 - for controlled variables 7-19
 - for defined variables 7-29
 - for parameter variables 7-20
 - for static variables 7-18
 - for strings 3-16
- external entry constant names
 - declaration of 8-37
- external procedures 5-20
 - binding 16-7
 - compiling 16-14
 - entering 16-14
 - linking 16-5
- external regions 7-4
- external scope 7-14
 - attribute 7-10

F

- f format item 14-30
- f indicator
 - fixed-point 3-40
 - floating-point 3-42
- factored declarations 6-6
- false value 2-3
- fatal conditions 13-13
- field in stream 14-28

- file
 - attribute 3-45
 - constant references 8-38
 - references
 - in condition references 14-6
 - to record files 15-4
 - values 2-4
- file descriptions
 - for record files 15-9
 - for streams 14-11
- file name 14-5, 15-4
- file option
 - in close statement 14-11, 15-10
 - in delete statement 15-12, 15-14
 - in get statement 14-13
 - in locate statement 15-20
 - in open statement 14-10, 15-8
 - in put statement 14-14
 - in read statement 15-11, 15-14, 15-16
 - in rewrite statement 15-12, 15-15
 - in write statement 15-11, 15-14
- file-state blocks
 - for record files 15-3
 - for streams 14-5
- files
 - attachment 16-9
 - record files 15-5
 - streams 14-7
 - opening
 - for streams 14-8
 - record files 15-6
 - references
 - for stream input/output 14-6
- filler storage 3-66
- filler zeros 3-8
- finish condition 13-20
- fixed
 - attribute 3-5
 - function 9-60
- fixed-point format items 14-30
- fixed-point picture attributes 3-22
- fixedoverflow condition
 - for arithmetic target 4-8
 - for conversion 4-20
- float
 - attribute 3-5
 - function 9-61
- floating-point format items 14-32
- floating-point indicator 3-41
- floating-point picture attributes 3-22
- floating-point values 3-6
- floor function 9-12
- flow of control 11-1
 - guidelines for 11-19
 - summary of 1-4
- format
 - attribute 3-45
- format constant names 8-35
- format lists 14-41
- format statement 14-15
 - in general recursion 12-40
 - sequential execution of 11-2
 - syntax diagram A-11
- FORTRAN control for do 11-13
- free statement
 - for based variables 7-22
 - for controlled variables 7-18
 - syntax diagram A-12
- from option
 - in rewrite statement 15-12, 15-15
 - in write statement 15-11, 15-14
 - omission of 15-20
- fully-qualified reference 6-17
- function references 12-14
 - interpretation of 12-15
 - result 12-16

G

- general
 - recursion 12-34
- generic entry names 12-27
- generic names
 - attributes for 6-23
- get statement 14-13
 - syntax diagram A-13
- goto statement 11-16
 - as procedure exit 12-11, 12-15
 - destination of 11-16
 - in general recursion 12-42
 - local attribute for 11-17
- goto statements
 - syntax diagram A-14
- guidelines
 - for %include macros 5-9
 - for abbreviations and defaults 3-8
 - for aggregates 3-60
 - for arguments 12-8
 - for arithmetic constants 8-32

- guidelines (cont)
 - for arithmetic data 3-14
 - for assignment statements 10-8
 - for built-in functions 8-45
 - for condition handling 13-17
 - for conditions
 - for conversion 4-22
 - for contextual declarations 6-11
 - for conversion functions 9-69
 - for data-directed input/output 14-20
 - for declare statements 6-7
 - for default statements 6-15
 - for edit-directed input/output 14-42
 - for flow of control 11-19
 - for identifiers 5-4
 - for implicit declarations 6-11
 - for like attribute 6-13
 - for list-directed input/output 14-25
 - for names 16-3
 - for pictured strings 3-44
 - for programmed functions 8-45
 - for programs 5-20
 - for storage class 7-32
 - for string data 3-19
 - for structures 3-50
 - for studying PL/I 1-12
 - for subscript list deletion 8-26
 - for variable references 8-29

- guidelines for
 - scope attributes 7-14

H

- hbound function 9-54
- high function 9-47
- hyperbolic functions 9-28

I

- identifiers
 - as lexemes 5-3
 - guidelines for 5-4
- if statement 11-3
 - consequences of 11-4
 - dangling else in 11-7
 - syntax diagram A-15
 - test is 11-4
 - within if statement 11-6
- imag
 - function 9-18
 - pseudo-variable 10-9
- imaginary outer block 6-4
- imaginary part 3-5, 10-9

- immediate containment
 - definition of 6-3
- implementation-dependent address functions 9-51
- implicit declaration 6-10
- implicit targets 4-4
- in option 7-22
- include macros 5-7
- inclusive-or operator 9-41
- increment of do group 11-13
- independent statement
 - as executable unit 11-5
- index function 9-34
- index of do group 11-10
- indexed sequential Multics file 15-3
- indicator 3-35
- infix operators 8-47
- infix sign operators 9-6
- initial attribute 7-10, 7-35
 - evaluation of 7-6
- initialization of do group 11-13
- input attribute
 - for record files 15-9
 - for streams 14-11
- input/output
 - conditions 13-4
 - summary of 1-5
- insertion-character indicator 3-38
- integers
 - implicit integer targets 4-5
- interactive streams
 - description of 14-11
- internal regions 7-4
- internal scope 7-13
 - attribute 7-10
- into option
 - in read statement 15-11, 15-14
- io_call command 16-10
- isub defined variables 7-31
- isubs
 - as lexemes 5-7

- isubs (cont)
 - restriction of 12-3
- iterated format lists 14-41
- iterative do group
 - with index 11-10
 - without index 11-9
- K
- k indicator 3-41
- key condition 15-25
- key option
 - in delete statement 15-12
 - in read statement 15-11, 15-16
 - in rewrite statement 15-12
- keyed attribute 15-9
- keyed record input/output 15-10
- keyfrom option
 - in locate statement 15-20
 - in write statement 15-11
- keyto option
 - in read statement 15-16
- keywords
 - in statements 5-14
 - versus names 5-3
- L
- label
 - attribute 3-45
- label constant names 8-35
- label prefix
 - for do statement 11-9
 - for end statement 11-9
 - purpose of 5-13
 - to declare a name 6-8
- language-defined conditions 13-2
 - references 13-6
- layout conventions 11-20
- lbound function 9-54
- left-adjusted string 14-30
- left-major order of arrays 3-57
- length function 9-40
- length of string 2-2, 3-16, 9-40
- letters 5-2
- level numbers 3-49
- level references 8-15
- level-one variable 3-49
- lexemes 5-2
 - %include macros 5-7
 - classification of 5-12
 - identifiers 5-3
 - isubs 5-7
 - literal constants 5-5
 - operators 5-6
 - pictures 5-7
 - punctuators 5-6
 - separators 5-10
- like attribute 6-11
- limit of do group 11-13
- line format item 14-40
- line number 14-5
- line option
 - in put statement 14-14
- line size 14-5
- linemark 14-1
- lineno function 9-71
- linking 16-5
- list option
 - in get statement 14-13
 - in put statement 14-14
- list-directed input/output 14-21
- listing segment 16-4
 - name 16-3
- literal constants
 - as lexemes 5-5
- local attribute 11-17
- local transfer of control 11-17
- locate statement 15-20
 - syntax diagram A-15
- location of variables 8-9
- locator
 - attributes 3-45
 - values 2-4
- locator qualifiers 8-21
- locator-qualified variable reference
 - locator-qualified deletion 8-28
- locator-qualified variable references 8-20

- locator-qualifiers
 - deletion of 8-28
- log function 9-22
- log10 function 9-22
- log2 function 9-22
- logical operators 9-41
- low function 9-47
- M
- major names 8-7
- major types 10-2
- major variable 3-49
- management class 7-10
- mantissa of float value 3-6, 3-7
- manuals on PL/I 1-12
- mathematical operations 9-19
- max function 9-11
- maximum length of string 3-16
- member references 8-15
- members of structures 3-49
- min function 9-11
- mod function 9-14
- mode attributes 3-5
 - for pictured storage 3-23
 - guidelines for choice of 3-14
- modules 16-11
 - for record files 15-5
 - for stream files 14-7
 - record_stream_ 16-12
 - syn_ 16-12
 - tty_ 16-11
 - vfile_ 16-11
- multi-valued functions 9-20
- Multics files 14-4, 15-2
- multiplication operator 9-7
- multiply function 9-10
- N
- name
 - segment 16-1
- name condition 14-46
- name-sequence
 - for declaration 6-16
 - for name reference 6-17
- names
 - component 16-2
 - correspondence with storage 7-14
 - deletion of 8-26
 - guidelines for 16-3
 - offset 16-2
 - segment
 - for compiler 16-3
 - special names 7-4
 - versus keywords 5-3
- natural logarithms 9-21
- nested expressions 8-2
- nesting
 - of blocks and groups 5-17
 - of do statements 11-16
 - of if statement 11-6
- next record indicator 15-2, 15-4
 - initialization of 15-8
- no-suppression digit indicator 3-30
- noncomputational values 2-5
- noniterative do group 11-6, 11-16
- nonlocal transfer of control 11-17
- nonnumeric indicator 3-43
- nonnumeric picture attributes 3-22
- nonstandard operations 9-3
- nonvarying
 - attribute 3-17
- normalized bounds of an array 3-56
- normalized structure levels 3-51
- not operator 9-41
- null function 9-50
- null record 15-2
- null statement
 - in if statement 11-8
 - syntax diagram A-16
- nullo function 9-50
- number system 3-7
- number-of-digits in precision 3-7

- numeric picture attributes 3-22
 - 0
- object segment 16-4
- occupation record 3-47
- occurrence of conditions 13-12
- offset
 - attribute 3-45
 - function 9-66
 - values 2-4
- offset names 16-2
- on statement 14-11
 - syntax diagram A-16
- on unit
 - defaults 13-15
- on units 13-13
 - establishing and reverting 13-10
 - for debugging 13-18
 - for file communication 13-18
 - in general recursion 12-41
 - restrictions 13-13
- onchar
 - function 9-78
 - pseudo-variable 10-13
- oncode function 9-76
- onfield function 9-77
- onfile function 9-79
 - use of 15-24
- onkey function 9-77
 - use of 15-24
- onloc function 9-76
- onsource
 - function 9-78
 - pseudo-variable 10-13
- open statement
 - for record files 15-8
 - for stream files 14-10
 - syntax diagram A-17
- opening mode 16-10
- operands 8-47
- operations 9-1
 - conventions for 9-3
 - mathematical operations 9-19
 - string operations 9-30
 - conventions for arithmetic operations 9-4
 - general rules 9-2
- operations (cont)
 - kinds of
 - address 9-48
 - area 9-48
 - arithmetic 9-3
 - array 9-53
 - conversion 9-56
 - mathematical 9-19
 - strings 9-30
 - system variable 9-70
 - nonstandard 9-3
 - summary of 1-4
- operator 9-7
- operator expressions 8-46
- operators
 - as lexemes 5-6
 - classification of 5-6
- optimization of expression evaluation 8-5
- options
 - general form 5-14
- options attribute 12-24
- or operator 9-41
- order of expression evaluation 8-5
- ordinary external regions 7-4
- output attribute
 - for record files 15-9
 - for streams 14-11
- output listing 16-4
- overflow condition
 - for arithmetic target 4-8
 - for conversion 4-21
- overlapping string targets 10-5

P

- p format item 14-35
- padding for strings 3-17, 9-31
- page format item 14-40
- page option
 - in put statement 14-14
- page size 14-5
- pagemark 14-1
- pageno
 - function 9-71
 - pseudo-variable 10-12

- parameter variables 7-20
 - attribute 7-10
 - guidelines for 7-34
- parameters 12-19
 - asterisk extent 12-7
 - restrictions on 12-19
- parent designator 12-2, 12-37
- parenthesized expressions 8-3
- parenthesized iterated list 14-23
- partial based variables 7-28
- partial segment names 5-7
- partially-qualified reference 6-18
- passing arguments 12-2
- pathname
 - absolute 16-1
 - relative 16-1
- performance measuring 16-18
- permanent internal regions 7-4
- phase of iterative do 11-11
- picture
 - attribute 3-21
 - complex 3-42
 - fixed-point 3-26, 3-29
 - floating-point 3-22, 3-27, 3-41
 - nonnumeric 3-22, 3-43
 - numeric 3-22
 - attributes
 - fixed-point 3-22
 - classification of pictures 3-22
 - guidelines for 3-44
 - indicators 3-21
 - arithmetic decimal point 3-39
 - decimal-point 3-32
 - digit 3-22
 - dollar 3-34
 - drifting-dollar 3-37
 - drifting-sign 3-36
 - floating-point 3-41
 - insertion characters 3-38
 - no-suppression digit 3-30
 - nonnumeric 3-43
 - scale-factor
 - fixed-point 3-40
 - floating-point 3-42
 - sign 3-33
 - zero-suppression 3-35
 - lexemes 5-7
 - storage 3-20
 - arithmetic assignments 3-27
 - arithmetic fetches 3-28
 - character-string assignments 3-25
 - character-string fetches 3-26
 - interpretation of 3-23
 - values 2-3
 - arithmetic 3-26
- picture (cont)
 - values
 - character strings 3-24
- picture format item 14-35
 - character 14-38
 - fixed-point 14-36
 - floating-point 14-38
 - use of 3-44
- p11 command 16-14
- pointer
 - attribute 3-45
 - function
 - nonstandard 9-52
 - standard 9-66
 - in recursion 12-36
 - values 2-4
- position attribute 7-10, 7-31
- power operator 9-9
- precedence for operators 8-48
- precision
 - attribute 3-7
 - guidelines for choice of 3-15
- precision function 9-64
- prefix of statement 5-13
 - condition prefix 13-7
 - procedure statement 12-18
- prefix operators 8-47
- prefix sign operators 9-6
- print attribute 14-11
- print_attach_table command 16-11
- priority for operators 8-48
- probe command 16-17
- procedure block
 - as program structure 5-18
 - external 5-20
 - in general recursion 12-39
- procedure statement 12-17
 - prefix of 12-18
 - syntax diagram A-18
- procedures 12-1, 12-17
 - activation 12-10
 - deactivation of 12-12
 - execution 12-11
 - exit from 12-11, 12-15
 - recursive 12-20, 12-28
- prod function 9-55
- produce block
 - sequential execution of 11-2

produce statement
 sequential execution of 11-2

product operator 9-7

profile command 16-18

program 5-20
 debugging 16-16
 execution 16-15
 guidelines for 5-20
 layout conventions 11-20
 monitoring 16-18
 structure of 1-3, 5-17
 syntax 5-1
 termination 16-16
 validity 1-10

program flow 11-1

programmed function references 8-39

programmer-defined conditions 13-5
 references 13-6

program_interrupt condition 14-5

promotion of aggregates 4-18

pseudo-variable
 in do statement 11-15

pseudo-variables
 in input/output statements 14-24
 interpretation of 10-9
 list of 10-4

punctuators
 as lexemes 5-6

put statement 14-14
 syntax diagram A-19

Q

qedx editor 16-14

quit condition 13-5

R

r format item 14-15

read statement
 for keyed input/output 15-11
 for sequential input/output 15-14
 syntax diagram A-20

real
 attribute 3-5, 3-23
 function 9-18, 9-59
 pseudo-variable 10-9
 values 2-1

real part 3-5, 10-9

record condition 15-25

record input/output 15-1
 attachment to Multics files 15-5
 based 15-16
 closing files 15-8
 conditions for 15-23
 data sets 15-2
 files 15-3
 keyed 15-10
 opening files 15-8
 operations 15-7
 sequential 15-13

record_stream_ I/O module 16-12

recursion 12-28
 activation indexes in 12-35
 activation variable reference 12-38
 begin block 12-40
 chained 12-32
 format statement 12-40
 general 12-34
 goto statement 12-42
 on units 12-41
 parent designators 12-2, 12-37
 pointers in 12-36
 procedure block 12-39
 statement address constant
 reference 12-39
 surface 12-34
 with arguments 12-31
 without arguments 12-29

recursive keyword 12-20

recursive procedures 12-20

refer option 7-24

reference
 built-in function reference 8-42
 destination of goto 11-16
 restriction 11-18
 locator-qualified 8-20
 programmed function reference 8-39
 shortened references 8-25
 simple 8-8
 structure-qualified 8-15
 subscripted 8-11
 to variable 8-7

references
 constant 8-35

rel function 9-52

related arithmetic types 3-26

related character types 3-24

relational operators 9-37
 for address values 9-48
 for arithmetic values 9-10

relative pathname 16-1

- release command 16-16
- remote format item 14-15
- remote format items 14-41
- repeat control for do 11-12
- replicators
 - in initial attribute 7-35
 - in picture 3-21
 - in string constants 8-32
- resolution of names 6-15
 - rules for 6-13
- resource reservation function 9-73
- return statement 12-22
 - as procedure exit 12-11, 12-15
 - assumed 11-2
 - syntax diagram A-21
- returns attribute 12-19
- reverse function 9-40
- revert statement 14-12
 - syntax diagram A-21
- reverting on units 13-10
- rewrite statement
 - for keyed input/output 15-12
 - for sequential input/output 15-15
 - syntax diagram A-21
- right-hand-side expression 10-4
- root block 6-4
- round function 9-13
- rounding
 - for conversion 4-8
 - for pictured strings 3-31

S

- s indicator
 - drifting 3-36
 - leftmost 3-33
- scalars
 - storage layout 3-67
 - values 2-5
- scale attributes 3-5
 - guidelines for choice of 3-15
- scale-factor in precision 3-8
- scale-factor indicator
 - fixed-point 3-40
 - floating-point 3-42
- scope
 - guidelines for 7-14
 - of condition prefix 13-8
- scope attributes 6-22
- scope of block 5-18
- scopes 7-13
- search function 9-43
- segment 16-1
 - directory 16-1
 - executable procedure 16-4
 - listing 16-4
 - object 16-4
- self-describing structures 15-19
- separation rules 5-10
- separator lexemes 5-10
- sequential attribute 15-9
- sequential execution 11-2
- sequential Multics files 15-2
- set option 7-22
 - in locate statement 15-20
 - in read statement 15-16
- side effects 8-6
- sign function 9-17
- sign indicator 3-33
- sign operators 9-6
- sign-manipulation functions 9-16
- signal statement 14-13
 - syntax diagram A-22
- signalling of conditions 13-13
- significant digits 3-8
- simple based variables 7-26
- simple declarations 6-4
- simple defined variables 7-30
- simple variable references 8-8
- sin function 9-24
- sind function 9-25
- single precision 3-7
- single-value control for do 11-11

- size condition
 - for arithmetic target 4-8
 - for conversion 4-20
 - for pictured strings 3-31
- size function 9-73
- skip format item 14-40
- skip option
 - in get statement 14-13
- snap keyword 13-11
- source segment
 - name 16-3
- spaces 5-2
 - as lexemes 5-10
- special array functions 9-55
- special characters 5-2
- special conversion functions 9-66
- special names 7-4
- special string target 10-6
- sqrt function 9-23
- stac function 9-74
- stack frame 12-10
 - in recursion 12-28
- stacking controlled variables 7-18
- start command 16-16
- statement
 - address values 2-3
 - classification of 5-16
 - parts of
 - options 5-14
 - attributes 5-15
 - body 5-14
 - clauses 5-14
 - keywords 5-14
 - prefix 5-13
- statement address constant references 12-39
- static variables 7-17
 - attribute 7-10
 - guidelines for 7-33
- statistical analysis functions 9-29
- status indicator 14-5, 15-4
- storage 3-1
- storage class attributes 6-22
 - guidelines for 7-32
- storage classes 7-15
- storage condition 7-39
- storage conditons 13-3
- storage layout 3-66
- storage limits 7-38
- storage management 7-1
 - conditions for 7-39
 - functions 9-72
 - fundamental principles 7-3
 - operations 7-6
- storage regions 7-4
 - capacity of 7-38
 - diagrams 7-5
- storage system 16-1
- storage types 3-2
 - addresses 3-44
 - aggregates 3-49
 - areas 3-46
 - arithmetic 3-4
 - conversion of 4-1
 - of expressions 8-3
 - ordinary strings 3-16
 - pictured strings 3-20
- storage unit
 - box 3-2
 - contents 3-2
- storage units 3-1
 - data frame 3-10
 - designator 3-2
 - examples of 3-10, 3-18
 - interpretation of 3-11
 - storage type 3-3
- stream attribute 14-10
- stream input/output 14-1
 - attachment to Multics files 14-7
 - closing files 14-9
 - conditions for 14-44
 - data sets 14-1
 - data-directed 14-16
 - edit-directed 14-26
 - files 14-4
 - list-directed 14-21
 - opening files 14-9
 - operations 14-8
 - statements 14-13
 - string option 14-43
- stream pointer 14-5
 - initialization of 14-10
- streams
 - input 14-3
 - output 14-3
 - pseudo 14-4

- string
 - function 9-67
 - pseudo-variable 10-11
- string format items 14-29
- string option 14-43
 - in get statement 14-14
- string options
 - in put statement 14-14
- string overlay based variables 7-27
- string overlay defined variables 7-31
- string-type attributes 3-16
- strings
 - assignment of 10-2
 - attributes 3-16, 3-21
 - constant literals 8-32
 - constants
 - as lexemes 5-5
 - implicit string targets 4-5
 - operations 9-30
 - ordinary 3-16
 - padding 3-17
 - pictured 3-20
 - values 2-2
- stringsize condition
 - for conversion 4-22
 - for string target 4-10
- structure
 - storage types 3-50
- structure declarations 6-5
- structure-qualified variable reference
 - name deletion 8-26
- structure-qualified variable
 - references 8-15
- structures 3-49
 - storage 3-51
 - storage layout 3-68
 - values 2-4
- subscripted variable reference
 - subscript-list deletion 8-25
- subscripted variable references 8-11
- subscriprange condition 8-11
 - in goto statement 11-17
- subscripts 8-11
 - deletion of 8-25
- subsets of PL/I
 - for business programming 1-7
 - for scientific programming 1-5
 - for system programming 1-8
- substr
 - function 9-33
 - pseudo-variable 10-10
- subtract function 9-10
- sum function 9-55
- sunh function 9-28
- supplementary storage 3-66
- switch
 - attachment
 - for record files 15-5
 - for stream files 14-7
 - I/O 16-10
 - name 16-10
 - opening
 - for record files 15-6
 - for stream files 14-8
 - standard 16-10
- switch goto statement 11-16
- syntactic validity 5-1
- syntax
 - general rules 5-1
 - of statements A-1
 - notation A-1
- syntax of statement
 - parts statement A-4
- syntax of statements
 - conventions A-5
- syn_ I/O module 16-12
- sysin stream file 14-12
- sysprint stream file 14-12
- system counter functions 9-71
- system keyword 13-11
- system variable 13-16
- system variable operations 9-70

T

- tan function 9-24
- tand function 9-25
- tanh function 9-28
- targets
 - for assignment 10-4
 - for conversion 4-2
 - implicit 4-4
- termination conditions 13-3

test
 in if statement 11-4
 in while option 11-9, 11-11, 11-12,
 11-13
 texts on PL/I 1-12
 then clause 11-3
 time function 9-71
 title option
 for record files 15-8
 for streams 14-10
 trace command 16-18
 transfer of control 11-9
 translate function 9-44
 transmission option
 in get statement 14-13
 in put statement 14-14
 transmit condition
 for record input/output 15-26
 for stream input/output 14-46
 trigonometric functions 9-24
 true value 2-3
 trunc function 9-12
 truncation functions 9-12
 truncation of arithmetic values
 for conversion 4-8
 tty_ I/O module 16-11

U

unaligned attribute 3-62
 unconnected aggregates
 restriction of 12-3
 undefined expression evaluation 8-6
 undefinedfile condition 14-10
 for record input/output 15-26
 for stream input/output 14-47
 underflow condition
 for conversion 4-21
 for arithmetic target 4-8
 unspec
 function 9-68
 pseudo-variable 10-12
 unstructured Multics files 14-4

update attribute
 for record files 15-9
 usage categories 7-12
 user_io switch 16-10

V

v indicator 3-39
 v. indicator 3-32
 valid function 9-69
 validity of programs 1-10
 values
 address 2-3
 aggregate 2-4
 area 2-4
 arithmetic 2-1
 array 2-4
 assignment of 10-1
 classification of 2-5
 conversion of 4-1
 kinds of values 2-1
 storage for 3-1
 string 2-2
 structure 2-4
 variability attributes 3-17
 variable names
 attributes for 6-21
 variable references 8-7
 locator qualified 8-20
 shortened references 8-25
 simple 8-8
 structure-qualified 8-15
 subscripted 8-11

variable storage 7-13
 attribute 7-10

variables 3-1

varying
 attribute 3-17

verify function 9-43

vfile_ I/O module 16-11

W

while option
 in FORTRAN control 11-13
 in repeat control 11-12
 in single-value control 11-11
 without index 11-9

working-directory 16-1

write statement
 for keyed input/output 15-11
 for sequential input/output 15-14
 syntax diagram A-22

X

x format item 14-39

x indicator 3-43

Y

y indicator 3-35

Z

z indicator 3-35

zero-suppression indicator 3-35

HONEYWELL INFORMATION SYSTEMS

Technical Publications Remarks Form

CUT ALONG L

TITLE **SERIES 60 (LEVEL 68) MULTICS PL/I
REFERENCE MANUAL
ADDENDUM A**

ORDER NO. **AM83A, REV. 0**

DATED **SEPTEMBER 1978**

ERRORS IN PUBLICATION

[Empty box for reporting errors in publication]

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

[Empty box for providing suggestions for improvement to publication]



Your comments will be promptly investigated by appropriate technical personnel and action will be taken as required. If you require a written reply, check here and furnish complete mailing address below.

FROM: NAME _____

DATE _____

TITLE _____

COMPANY _____

ADDRESS _____

PLEASE FOLD AND TAPE –

NOTE: U. S. Postal Service will not deliver stapled forms

FIRST CLASS
PERMIT NO. 39531
WALTHAM, MA
02154

Business Reply Mail
Postage Stamp Not Necessary if Mailed in the United States

Postage Will Be Paid By:

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTENTION: PUBLICATIONS, MS 486

Honeywell

CUT ALONG

MULTICS PL/I REFERENCE MANUAL ADDENDUM A

SUBJECT

Additions and Changes to the Reference Manual for Multics PL/I

SPECIAL INSTRUCTIONS

This is the first addendum to the Multics PL/I Reference Manual, Revision 0, dated June 1976.

Insert the attached pages according to the collating instructions on the back of this cover. Throughout the manual, change bars in the margins indicate technical additions and changes; asterisks denote deletions. These changes will be incorporated in the next revision of the manual. The index has not been updated but all single quote items have been merged with the regular index. Also, the items in Section 13, which appeared as being in Section 14, have been corrected. The changed items in this addendum are:

begin statement	procedure statement
on statement	translate, search, verify built-ins

The new items are:

acos, asin built-ins	maxlength built-in
clock, vclock built-ins	octal, hexadecimal constants
collate9, high9 built-ins	options(constant)
currentsize built-in	stacq built-in
fixed, scaled constants	read statement
ltrim, rtrim built-ins	write statement

The following items are documented in this addendum but will not be available until MR7.0. They are:

4-bit decimal	stackbaseptr
codeptr	stackframeptr
environmentptr	stop statement
nonvar keyword	unsigned binary
options(main)	

Work will begin in January on a revision to this manual. This will be the first complete revision since the manual was originally published, so we are interested in getting information about the things you feel the book does or does not do and an explanation of what these things are. All of your comments will be reviewed and, depending on the time available, as many as possible will be incorporated in the manual. Those that are not done at this time will be kept for future revisions or addenda. If it is decided that a suggestion cannot be used, we will write a response to that suggestion explaining why it was not used.

A questionnaire precedes the standard user remark form on the last page of this addendum.

Note:

Insert this cover after the manual cover to indicate the updating of this document with Addendum A.

SOFTWARE SUPPORTED

Multics Software Release 7.0

ORDER NUMBER

AM83A, Rev. 0

September 1978

36668
2. 25C83
Printed in U.S.A.

Honeywell

COLLATING INSTRUCTIONS

To update this manual, remove old pages and insert new pages as follows:

<u>Remove</u>	<u>Insert</u>
title page, preface	title page, preface
table of contents	v through xx
1-1, 1-2	1-1, 1-2
1-9, 1-10	1-9, 1-10
1-13, 1-14	1-13, 1-14
3-17, 3-18	3-17, 3-18
3-33 through 3-38	3-33 through 3-38
3-55, 3-56	3-55, 3-56
5-7 through 5-10	5-7 through 5-10
5-15, 5-16	5-15, 5-16
7-19 through 7-22	7-19 through 7-22
7-37 through 7-40	7-37 through 7-40
8-17 through 8-20	8-17 through 8-20
8-31 through 8-34	8-31 through 8-34
8-39, 8-40	8-39, 8-40
8-43, 8-44	8-43, 8-44
9-3, 9-4	9-3, 9-4
9-17, 9-18	9-17, 9-18
9-23, 9-24	9-23, 9-24 9-24.1, blank
9-29, 9-30	9-29, 9-30
9-37 through 9-52	9-37 through 9-40 9-40.1, blank 9-41 through 9-52 9-52.1, 9-52.2
9-57, 9-58	9-57, 9-58
9-71 through 9-74	9-71 through 9-74 9-74.1, blank

Remove

9-57, 9-58
9-71 through 9-74

10-13, 10-14
11-11, 11-12
12-5, 12-6
12-9, 12-10
12-17, 12-18
12-31, 12-32
13-11 through 13-14
14-1 through 14-4
14-35 through 14-38
14-43, 14-44

15-5, 15-6
15-21 through 15-26
16-9, 16-10
16-13, 16-14
16-17 through 16-20
A-5 through A-8
A-15 through A-18
A-21, A-22

i-1 through i-19

Insert

9-57, 9-58
9-71 through 9-74
9-74.1, blank

10-13, 10-14
11-11, 11-12
12-5, 12-6
12-9, 12-10
12-17, 12-18
12-31, 12-32
13-11 through 13-14
14-1 through 14-4
14-35 through 14-38
14-43, 14-44
14-44.1, blank

15-5, 15-6
15-21 through 15-24
16-9, 16-10
16-13, 16-14
16-17 through 16-20
A-5 through A-8
A-15 through A-18
A-21, A-22

B-1, B-2

i-1 through i-19
questionnaire

Honeywell

Honeywell Information Systems

In the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154

In Canada: 2025 Sheppard Avenue East, Willowdale, Ontario M2J 1W5

In the U.K.: Great West Road, Brentford, Middlesex TW8 9DH

In Australia: 124 Walker Street, North Sydney, N.S.W. 2060

In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.