

**IBM PL/I for MVS & VM  
Programming Guide  
Release 1.1**

Document Number SC26-3113-01

**Note!**

Before using this information and the product it supports, be sure to read the general information under "Notices" on page xii.

**Second Edition (June 1995)**

This edition applies to Version 1 Release 1.1 of IBM PL/I for MVS & VM (named IBM SAA AD/Cycle PL/I MVS & VM for Release 1.0), 5688-235, and to any subsequent releases until otherwise indicated in new editions or technical newsletters. Make sure you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

A form for readers' comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation, Department J58  
P.O. Box 49023  
San Jose, CA, 95161-9023  
United States of America

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© **Copyright International Business Machines Corporation 1964, 1995. All rights reserved.**

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

---

# Contents

<b>Notices</b> . . . . .	xii
Programming Interface Information . . . . .	xii
Trademarks . . . . .	xii

---

## Part 1. Introduction . . . . . xiii

<b>About this book</b> . . . . .	xiv
Run-time Environment for PL/I for MVS & VM . . . . .	xiv
Debugging Facility for PL/I for MVS & VM . . . . .	xiv
Using Your Documentation . . . . .	xiv
Where to Look for More Information . . . . .	xiv
What Is New in PL/I for MVS & VM . . . . .	xv
Notation Conventions Used in this Book . . . . .	xvii
Conventions Used . . . . .	xviii
How to Read the Syntax Notation . . . . .	xviii
How to Read the Notational Symbols . . . . .	xix

---

## Part 2. Compiling your program . . . . . 1

<b>Chapter 1. Using Compile-Time Options and Facilities</b> . . . . .	5
Compile-Time Option Descriptions . . . . .	5
AGGREGATE . . . . .	8
ATTRIBUTES . . . . .	8
CMPAT . . . . .	8
COMPILE . . . . .	10
CONTROL . . . . .	10
DECK . . . . .	10
ESD . . . . .	11
FLAG . . . . .	11
GONUMBER . . . . .	11
GOSTMT . . . . .	12
GRAPHIC . . . . .	12
IMPRECISE . . . . .	12
INCLUDE . . . . .	13
INSOURCE . . . . .	13
INTERRUPT . . . . .	13
LANGLVL . . . . .	14
LINECOUNT . . . . .	14
LIST . . . . .	14
LMESSAGE . . . . .	15
MACRO . . . . .	15
MAP . . . . .	15
MARGINI . . . . .	16
MARGINS . . . . .	16
MDECK . . . . .	17
NAME . . . . .	17
NEST . . . . .	18
NOT . . . . .	18
NUMBER . . . . .	18

OBJECT	19
OFFSET	19
OPTIMIZE	20
OPTIONS	20
OR	21
SEQUENCE	21
SIZE	22
SMESSAGE	23
SOURCE	23
STMT	23
STORAGE	23
SYNTAX	24
SYSTEM	24
TERMINAL	25
TEST	26
XREF	27
Input Record Formats	28
Specifying Options in the %PROCESS or *PROCESS statements	28
Using the Preprocessor	29
Invoking the Preprocessor	29
Using the %INCLUDE Statement	30
Using the PL/I Preprocessor in Program Testing	31
Using % Statements	32
Invoking the Compiler from an Assembler Routine	32
Option List	33
DDNAME List	33
Page Number	33
Using the Compiler Listing	33
Heading Information	34
Options Used for the Compilation	34
Preprocessor Input	34
SOURCE Program	34
Statement Nesting Level	36
ATTRIBUTE and Cross-Reference Table	36
Aggregate Length Table	38
Storage Requirements	39
Statement Offset Addresses	40
External Symbol Dictionary	41
Static Internal Storage Map	43
Object Listing	44
Messages	44
Return Codes	45
<b>Chapter 2. Using PL/I Cataloged Procedures under MVS</b>	<b>46</b>
IBM-Supplied Cataloged Procedures	46
Compile Only (IEL1C)	47
Compile and Link-Edit (IEL1CL)	48
Compile, Link-Edit, and Run (IEL1CLG)	50
Compile, Load and Run (IEL1CG)	51
Invoking a Cataloged Procedure	52
Specifying Multiple Invocations	52
Link-Editing Multitasking Programs	53
Modifying the PL/I Cataloged Procedures	54
EXEC Statement	54

DD Statement	55
<b>Chapter 3. Compiling under MVS</b>	<b>56</b>
Invoking the Compiler under TSO	56
Allocating Data Sets	57
Using the PLI Command	59
Compiler Listings	62
Running Jobs in a Background Region	63
Using JCL during Compilation	64
EXEC Statement	64
DD Statements for the Standard Data Sets	64
Temporary Workfile (SYSUT1)	66
Listing (SYSPRINT)	67
Source Statement Library (SYSLIB)	67
Example of Compiler JCL	67
Specifying Options	68
Specifying Options in the EXEC Statement	68
Compiling Multiple Procedures in a Single Job Step	69
SIZE Option	69
NAME Option	69
Return Codes in Batched Compilation	70
Job Control Language for Batched Processing	70
Examples of Batched Compilations	71
Correcting Compiler-Detected Errors	71
The PL/I Compiler and MVS/ESA	71
Compiling for CICS	72
<b>Chapter 4. Compiling under VM</b>	<b>73</b>
Using the PLIOPT Command	73
Compiler Output and Its Destination	73
Compile-Time Options	74
Files Used by the Compiler	74
PLIOPT Command Options	75
PL/I Batched Compilation	78
Correcting Compiler-Detected Errors	78
<b>Chapter 5. Link-Editing and Running</b>	<b>80</b>
Selecting Math Results at Link-Edit Time	80
VM Run-Time Considerations	80
Separately Compiled PL/I MAIN Programs	81
Using Data Sets and Files	81
Restrictions Using PL/I under VM	85
PL/I Conventions under VM	86
MVS Run-Time Considerations	89
Formatting Conventions for PRINT Files	89
Changing the Format on PRINT Files	89
Automatic Prompting	90
Punctuating Long Input Lines	91
Punctuating GET LIST and GET DATA Statements	91
ENDFILE	92
SYSPRINT Considerations	92

<b>Part 3. Using I/O facilities</b> .....	95
<b>Chapter 6. Using Data Sets and Files</b> .....	99
Associating Data Sets with Files .....	99
Associating Several Files with One Data Set .....	101
Associating Several Data Sets with One File .....	102
Concatenating Several Data Sets .....	102
Establishing Data Set Characteristics .....	102
Blocks and Records .....	103
Record Formats .....	103
Data Set Organization .....	106
Labels .....	107
Data Definition (DD) Statement .....	107
Associating PL/I Files with Data Sets .....	109
Specifying Characteristics in the ENVIRONMENT Attribute .....	110
Data Set Types Used by PL/I Record I/O .....	121
<b>Chapter 7. Using Libraries</b> .....	123
Types of libraries .....	123
How to Use a Library .....	123
Creating a Library .....	124
SPACE Parameter .....	124
Creating and Updating a Library Member .....	125
Examples .....	125
Extracting Information from a Library Directory .....	128
<b>Chapter 8. Defining and Using Consecutive Data Sets</b> .....	129
Using Stream-Oriented Data Transmission .....	129
Defining Files Using Stream I/O .....	130
Specifying ENVIRONMENT Options .....	130
Creating a Data Set with Stream I/O .....	132
Accessing a Data Set with Stream I/O .....	136
Using PRINT Files with Stream I/O .....	138
Using SYSIN and SYSPRINT Files .....	142
Controlling Input from the Terminal .....	143
Controlling Output to the Terminal .....	145
Example of an Interactive Program .....	146
Using Record-Oriented Data Transmission .....	149
Defining Files Using Record I/O .....	150
Specifying ENVIRONMENT Options .....	150
Creating a Data Set with Record I/O .....	156
Accessing and Updating a Data Set with Record I/O .....	157
<b>Chapter 9. Defining and Using Indexed Data Sets</b> .....	163
Indexed Organization .....	163
Using keys .....	163
Using Indexes .....	166
Defining Files for an Indexed Data Set .....	169
Specifying ENVIRONMENT Options .....	169
Creating an Indexed Data Set .....	172
Essential Information .....	172
Name of the Data Set .....	175
Record Format and Keys .....	175

Overflow Area	177
Master Index	178
Accessing and Updating an Indexed Data Set	179
Using Sequential Access	180
Using Direct Access	181
Reorganizing an Indexed Data Set	184
<b>Chapter 10. Defining and Using Regional Data Sets</b>	185
Defining Files for a Regional Data Set	188
Specifying ENVIRONMENT Options	188
Using Keys with REGIONAL Data Sets	190
Using REGIONAL(1) Data Sets	190
Creating a REGIONAL(1) Data Set	191
Accessing and Updating a REGIONAL(1) Data Set	192
Using REGIONAL(2) Data Sets	195
Using Keys for REGIONAL(2) and (3) Data Sets	195
Creating a REGIONAL(2) Data Set	197
Accessing and Updating a REGIONAL(2) Data Set	198
Using REGIONAL(3) Data Sets	202
Creating a REGIONAL(3) Data Set	202
Accessing and Updating a REGIONAL(3) Data Set	204
Essential Information for Creating and Accessing Regional Data Sets	208
<b>Chapter 11. Defining and Using VSAM Data Sets</b>	211
Using VSAM Data Sets	211
How to Run a Program with VSAM Data Sets	211
VSAM Organization	212
Keys for VSAM Data Sets	215
Choosing a Data Set Type	216
Defining Files for VSAM Data Sets	218
Specifying ENVIRONMENT Options	219
Performance Options	223
Defining Files for Alternate Index Paths	223
Using Files Defined for non-VSAM Data Sets	224
CONSECUTIVE Files	224
INDEXED Files	224
Using the VSAM Compatibility Interface	225
Adapting Existing Programs for VSAM	225
Using Several Files in One VSAM Data Set	226
Using Shared Data Sets	227
Defining VSAM Data Sets	227
Entry-Sequenced Data Sets	228
Loading an ESDS	229
Using a SEQUENTIAL File to Access an ESDS	229
Key-Sequenced and Indexed Entry-Sequenced Data Sets	232
Loading a KSDS or Indexed ESDS	234
Using a SEQUENTIAL File to Access a KSDS or Indexed ESDS	236
Using a DIRECT File to Access a KSDS or Indexed ESDS	236
Alternate Indexes for KSDSs or Indexed ESDSs	239
Relative-Record Data Sets	247
Loading an RRDS	249
Using a SEQUENTIAL File to Access an RRDS	251
Using a DIRECT File to Access an RRDS	252

<b>Chapter 12. Defining and Using Teleprocessing Data Sets</b> . . . . .	255
Message Control Program (MCP) . . . . .	255
TCAM Message Processing Program (TCAM MPP) . . . . .	256
Teleprocessing Organization . . . . .	256
Essential Information . . . . .	257
Defining Files for a Teleprocessing Data Set . . . . .	257
Specifying ENVIRONMENT Options . . . . .	257
Writing a TCAM Message Processing Program (TCAM MPP) . . . . .	258
Handling PL/I Conditions . . . . .	260
TCAM MPP Example . . . . .	261

---

**Part 4. Improving your program** . . . . . 263

<b>Chapter 13. Examining and Tuning Compiled Modules</b> . . . . .	265
Activating Hooks in Your Compiled Program Using IBMBHKS . . . . .	265
The IBMBHKS Programming Interface . . . . .	265
Obtaining Static Information about Compiled Modules Using IBMBSIR . . . . .	266
The IBMBSIR Programming Interface . . . . .	267
Obtaining Static Information as Hooks Are Executed Using IBMBHIR . . . . .	271
The IBMBHIR Programming Interface . . . . .	271
Examining Your Program's Run-Time Behavior . . . . .	272
Sample Facility 1: Examining Code Coverage . . . . .	272
Sample Facility 2: Performing Function Tracing . . . . .	284
Sample Facility 3: Analyzing CPU-Time Usage . . . . .	288
<b>Chapter 14. Efficient Programming</b> . . . . .	305
Efficient Performance . . . . .	305
Tuning a PL/I Program . . . . .	305
Tuning a Program for a Virtual Storage System . . . . .	307
Global Optimization Features . . . . .	308
Expressions . . . . .	309
Loops . . . . .	312
Arrays and Structures . . . . .	313
In-Line Code . . . . .	314
Key handling for REGIONAL data sets . . . . .	314
Matching Format Lists with Data Lists . . . . .	315
Run-time Library Routines . . . . .	315
Use of Registers . . . . .	315
Program Constructs that Inhibit Optimization . . . . .	315
Global Optimization of Variables . . . . .	316
ORDER and REORDER Options . . . . .	316
Common Expression Elimination . . . . .	318
Condition Handling for Programs with Common Expression Elimination . . . . .	320
Transfer of Invariant Expressions . . . . .	321
Redundant Expression Elimination . . . . .	322
Other Optimization Features . . . . .	322
Assignments and Initialization . . . . .	323
Notes about Data Elements . . . . .	323
Notes about Expressions and References . . . . .	326
Notes about Data Conversion . . . . .	329
Notes about Program Organization . . . . .	331
Notes about Recognition of Names . . . . .	332
Notes about Storage Control . . . . .	332

Notes about Statements . . . . .	334
Notes about Subroutines and Functions . . . . .	338
Notes about Built-In Functions and Pseudovariables . . . . .	338
Notes about Input and Output . . . . .	339
Notes about Record-Oriented Data Transmission . . . . .	340
Notes about Stream-Oriented Data Transmission . . . . .	341
Notes about Picture Specification Characters . . . . .	343
Notes about Condition Handling . . . . .	344
Notes about multitasking . . . . .	345

---

**Part 5. Using interfaces to other products . . . . . 347**

<b>Chapter 15. Using the Sort Program . . . . .</b>	<b>348</b>
Preparing to Use Sort . . . . .	348
Choosing the Type of Sort . . . . .	349
Specifying the Sorting Field . . . . .	352
Specifying the Records to be Sorted . . . . .	354
Determining Storage Needed for Sort . . . . .	355
Calling the Sort Program . . . . .	355
Determining Whether the Sort Was Successful . . . . .	358
Establishing Data Sets for Sort . . . . .	358
Sort Data Input and Output . . . . .	360
Data Input and Output Handling Routines . . . . .	360
E15 — Input Handling Routine (Sort Exit E15) . . . . .	361
E35 — Output Handling Routine (Sort Exit E35) . . . . .	364
Calling PLISRTA Example . . . . .	365
Calling PLISRTB Example . . . . .	366
Calling PLISRTC Example . . . . .	367
Calling PLISRTD Example . . . . .	368
Sorting Variable-Length Records Example . . . . .	369

---

**Part 6. Specialized programming tasks . . . . . 371**

<b>Chapter 16. Parameter Passing and Data Descriptors . . . . .</b>	<b>373</b>
PL/I Parameter Passing Conventions . . . . .	373
Passing Assembler Parameters . . . . .	374
Passing MAIN Procedure Parameters . . . . .	376
Options BYVALUE . . . . .	378
Descriptors and Locators . . . . .	380
Aggregate Locator . . . . .	381
Area Locator/Descriptor . . . . .	381
Array Descriptor . . . . .	382
String Locator/Descriptor . . . . .	383
Structure Descriptor . . . . .	384
Arrays of Structures and Structures of Arrays . . . . .	385
<b>Chapter 17. Using PLIDUMP . . . . .</b>	<b>386</b>
PLIDUMP Usage Notes . . . . .	387
<b>Chapter 18. Retaining the Run-Time Environment for Multiple     Invocations . . . . .</b>	<b>389</b>
Preinitializable Programs . . . . .	389

The Interface for Preinitializable Programs	390
Preinitializing a PL/I Program	393
Invoking an Alternative MAIN Routine	398
Using the Service Vector and Associated Routines	402
User Exits in Preinitializable Programs	419
The SYSTEM Option in Preinitializable Programs	419
Calling a Preinitializable Program under VM	419
Calling a Preinitializable Program under MVS	419
Establishing an Language Environment for MVS & VM-Enabled Assembler Routine as the MAIN Procedure	421
Retaining the Run-Time Environment Using Language Environment for MVS & VM-Enabled Assembler as MAIN	421
<b>Chapter 19. Multitasking in PL/I</b>	422
PL/I Multitasking Facilities	422
Creating PL/I Tasks	423
The TASK Option of the CALL Statement	423
The EVENT Option of the CALL Statement	423
The PRIORITY Option of the CALL Statement	424
Synchronization and Coordination of Tasks	424
Sharing Data between Tasks	425
Sharing Files between Tasks	425
Producing More Reliable Tasking Programs	426
Terminating PL/I Tasks	426
Dispatching Priority of Tasks	427
Running Tasking Programs	428
Sample Program 1: Multiple Independent Processes	428
Multiple Independent Processes: Nontasking Version	429
Multiple Independent Processes: Tasking Version	430
Sample Program 2: Multiple Independent Computations	432
Multiple Independent Computations: Nontasking Version	433
Multiple Independent Computations: Tasking Version	434
<b>Chapter 20. Interrupts and Attention Processing</b>	436
Using ATTENTION ON-Units	437
Interaction with a Debugging Tool	437
<b>Chapter 21. Using the Checkpoint/Restart Facility</b>	438
Requesting a Checkpoint Record	439
Defining the Checkpoint Data Set	440
Requesting a Restart	440
Automatic Restart after a System Failure	440
Automatic Restart within a Program	441
Getting a Deferred Restart	441
Modifying Checkpoint/Restart Activity	441
<b>Part 7. Appendix</b>	443
<b>Appendix. Sample Program IBMLSO1</b>	444
<b>Bibliography</b>	500
PL/I for MVS & VM Publications	500
Language Environment for MVS & VM Publications	500

PL/I for OS/2 Publications . . . . .	500
CoOperative Development Environment/370 . . . . .	500
IBM Debug Tool . . . . .	500
Softcopy Publications . . . . .	500
Other Books You Might Need . . . . .	500
<b>Glossary</b> . . . . .	<b>502</b>
<b>Index</b> . . . . .	<b>516</b>

---

## Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service can be used. Any functionally equivalent product, program, or service that does not infringe any of the intellectual property rights of IBM might be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM might have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594, U.S.A.

---

## Programming Interface Information

This book is intended to help the customer write programs using IBM PL/I for MVS & VM. This book documents General-use Programming Interface and Associated Guidance Information provided by IBM PL/I for MVS & VM.

General-use programming interfaces allow the customer to write programs that obtain the services of IBM PL/I for MVS & VM.

### Macros for Customer Use

IBM PL/I for MVS & VM provides no macros that allow a customer installation to write programs that use the services of IBM PL/I for MVS & VM.

**Warning:** Do not use as programming interfaces any IBM PL/I for MVS & VM macros.

---

## Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

3090	MVS/ESA
AD/Cycle	MVS/SP
CICS	MVS/XA
VM	OS/2
COBOL/370	Presentation Manager
DB2	SAA
IBM	System/390
Language Environment	VM/ESA
MVS/DFP	VM/XA

---

## Part 1. Introduction

<b>About this book</b> . . . . .	xiv
Run-time Environment for PL/I for MVS & VM . . . . .	xiv
Debugging Facility for PL/I for MVS & VM . . . . .	xiv
Using Your Documentation . . . . .	xiv
Where to Look for More Information . . . . .	xiv
PL/I Information . . . . .	xiv
Language Environment Information . . . . .	xv
What Is New in PL/I for MVS & VM . . . . .	xv
Notation Conventions Used in this Book . . . . .	xvii
Conventions Used . . . . .	xviii
How to Read the Syntax Notation . . . . .	xviii
How to Read the Notational Symbols . . . . .	xix
Example of Notation . . . . .	xx

---

## About this book

This book is for PL/I programmers and system programmers. It helps you understand how to use PL/I for MVS & VM to compile PL/I programs. It also describes the operating system features that you might need to optimize program performance or handle errors.

---

### Run-time Environment for PL/I for MVS & VM

PL/I for MVS & VM uses Language Environment as its run-time environment. It conforms to Language Environment architecture and can share the run-time environment with other Language Environment-conforming languages.

Language Environment provides a common set of run-time options and callable services. It also improves interlanguage communication (ILC) between high-level languages (HLL) and assembler by eliminating language-specific initialization and termination on each ILC invocation.

---

### Debugging Facility for PL/I for MVS & VM

PL/I for MVS & VM uses the IBM Debug Tool as its debugging facility on MVS and VM. Debug Tool utilizes the common run-time environment, Language Environment, to provide the ILC debugging capability among Language Environment-conforming languages. It also provides debugging capability under CICS. Debug Tool is compatible with INSPECT for C/370 and PL/I debugging facility. It provides equivalent functions that PLITEST as for OS PL/I Debug Tool provides the compatibility support for OS PL/I Version 2 and the same level of toleration that PLITEST used to provide for OS PL/I Version 1.

---

### Using Your Documentation

The publications provided with PL/I for MVS & VM are designed to help you do PL/I programming under MVS or VM. Each publication helps you perform a different task.

### Where to Look for More Information

The following tables show you how to use the publications you receive with PL/I for MVS & VM and Language Environment. You'll want to know information about both your compiler and run-time environment. For the complete titles and order numbers of these and other related publications, such as the IBM Debug Tool, see the "Bibliography" on page 500.

#### PL/I Information

*Table 1 (Page 1 of 2). How to Use Publications You Receive with PL/I for MVS & VM*

To...	Use...
Understand warranty information	Licensed Programming Specifications
Plan for, install, customize, and maintain PL/I	Installation and Customization under MVS Program Directory under VM

Table 1 (Page 2 of 2). How to Use Publications You Receive with PL/I for MVS & VM

To...	Use...
Understand compiler and run-time changes and adapt programs to PL/I and Language Environment	Compiler and Run-Time Migration Guide
Prepare and test your programs and get details on compiler options	Programming Guide
Get details on PL/I syntax and specifications of language elements	Language Reference Reference Summary
Diagnose compiler problems and report them to IBM	Diagnosis Guide
Get details on compile-time messages	Compile-Time Messages and Codes

## Language Environment Information

Table 2. How to Use Publications You Receive with Language Environment for MVS & VM

To...	Use...
Evaluate Language Environment	Fact Sheet Concepts Guide
Understand warranty information	Licensed Program Specifications
Understand the Language Environment program models and concepts	Concepts Guide Programming Guide
Plan for, install, customize, and maintain Language Environment on MVS	Installation and Customization under MVS Program Directory under VM
Migrate applications to Language Environment	Run-Time Migration Guide Your language migration guide
Find syntax for run-time options and callable services	Programming Reference
Develop your Language Environment-conforming applications	Programming Guide and your language programming guide
Find syntax for run-time options and callable services	Programming Reference
Develop interlanguage communication (ILC) applications	Writing Interlanguage Communication Applications
Debug your Language Environment-conforming application and get details on run-time messages	Debugging Guide and Run-Time Messages
Diagnose problems with Language Environment	Debugging Guide and Run-Time Messages
Find information in the Language Environment library quickly	Master Index

## What Is New in PL/I for MVS & VM

PL/I for MVS & VM enables you to integrate your PL/I applications into Language Environment for MVS & VM. In addition to PL/I's already impressive features, you gain access to Language Environment's rich set of library routines and enhanced interlanguage communication (ILC) with COBOL for MVS & VM, C/370, and C/C++ for MVS/ESA. Differences between OS PL/I and Language Environment's support of PL/I for MVS & VM are described in the *PL/I for MVS & VM Compiler and Run-Time Migration Guide*.

PL/I for MVS & VM Release 1.1 provides the following enhancements:

- Language Environment support of the PL/I multitasking facility
- Language Environment compatibility support for the following OS PL/I features:

- OS PL/I PLICALLA entry support extended to OS PL/I applications that have been recompiled with PL/I for MVS & VM
- OS PL/I PLICALLB entry support with some differences in handling storage
- Object and/or load module support for OS PL/I expanded to Version 1 Release 3.0-5.1 with some restrictions
- Support for OS PL/I load modules invoking PLISRTx
- Expanded support and rules for OS PL/I Shared Library
- OS PL/I coexistence with Language Environment
- Enhanced SYSPRINT support
- OS PL/I-Assembler clarifications
- Compatibility for location of heap storage
- Help to relink your object and load modules with Language Environment
- Help to relink your OS PL/I-COBOL ILC load modules with Language Environment
- Help to relink your OS PL/I load modules using PLISRTx with Language Environment
- Help to relink your OS PL/I Shared Library
- Enhanced ILC support for PL/I and C/370

Release 1.0 provided the following functions:

- IBM Language Environment for MVS & VM support including:
  - ILC support with COBOL for MVS & VM and C/370.
    - Object code produced by PL/I for MVS & VM Version 1 Release 1
    - Object code produced by all releases of OS PL/I Version 2 and Version 1 Release 5.1
    - Object code produced by LE/370-conforming compilers (all releases)
    - PL/I load modules can be fetched by COBOL/370 and C/370 load modules
    - Load modules from other LE/370 Version 1 Release 1 and Release 1.1 conforming languages. Some load module support for non-LE/370-conforming languages See the *PL/I for MVS & VM Compiler and Run-Time Migration Guide* for details.
    - Object code from VS COBOL II Version 1 Release 3 and C/370 Version 1 and Version 2 as provided by each respective Language Environment-conforming products)
- Note:** PL/I for MVS & VM does not support ILC with FORTRAN or OS/VS COBOL.
- Support for PL/I and C/370 ILC is enhanced.
  - Pointer data type now supports the null value used by C/370 and programs via the SYSNULL built-in function.
- Under VM, the source listings for PL/I compilations can now be directed to the printer by modifying an IBM-supplied EXEC.

- CEESTART is the entry point for all environments (including CICS).
  - Support for FETCH in CICS and VM.
  - Procedure OPTIONS option FETCHABLE can be used to specify the procedure that gets control within a fetched load module.
  - Implicit LE/370 enclave is created if the PL/I load module containing a MAIN procedure is fetched or is dynamically called.
  - CEETDLI is supported in addition to PLITDLI, ASMTDLI, and EXEC DLI.
  - By default, only user-generated output is written to SYSPRINT. All run-time generated messages are written to MSGFILE.
  - Automatic storage can now be above the 16-megabyte line.
  - All PL/I MVS & VM Version 1 Release 1 resident library routines are in a LIBPACK, and packaged with LE/370. The transient routines remain transient and are not packaged as part of the LIBPACK.
  - At link-edit time, you have the option of getting math results that are compatible with LE/370 or with OS PL/I.
- Support for DFP Version 3 system-determined blocksize.
  - DATETIME and TIME return milliseconds in all environments, including VM and CICS.
  - VM terminal I/O is unblocked and immediate.
  - ERROR conditions now get control of all system abends. The PL/I message is issued only if there is no ERROR on-unit or if the ERROR on-unit does not recover from the condition via a GOTO.
  - Selected items from OS/2 PL/I are implemented to allow better coexistence with PL/I Package/2.
    - Limited support of OPTIONS(BYVALUE and BYADDR)
    - Limited support of EXTERNAL(environment-name) allowing alternate external name
    - Limited support of OPTIONAL arguments/parameters
    - Support for %PROCESS statement
    - NOT and OR compiler options
  - Installation enhancements are provided to ease product installation and migration.

**Note:** You cannot use INSPECT for C/370 and PL/I or PLITEST with PL/I for MVS & VM

---

## Notation Conventions Used in this Book

This book uses the conventions, diagramming techniques, and notation described in “Conventions Used” on page xviii and “How to Read the Notational Symbols” on page xix to illustrate PL/I and non-PL/I programming syntax.

## Conventions Used

Some of the programming syntax in this book uses type fonts to denote different elements:

- Items shown in UPPERCASE letters indicate key elements that must be typed exactly as shown.
- Items shown in lowercase letters indicate user-supplied variables for which you must substitute appropriate names or values. The variables begin with a letter and can include hyphens, numbers, or the underscore character (\_).
- The term *digit* indicates that a digit (0 through 9) should be substituted.
- The term *do-group* indicates that a do-group should be substituted.
- Underlined items indicate default options.
- Examples are shown in monospace type.
- Unless otherwise indicated, separate repeatable items from each other by one or more blanks.

**Note:** Any symbols shown that are not purely notational, as described in “How to Read the Notational Symbols” on page xix, are part of the programming syntax itself.

For an example of programming syntax that follows these conventions, see “Example of Notation” on page xx.

## How to Read the Syntax Notation

Throughout this book, syntax is described using the following structure:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line. The following table shows the meaning of symbols at the beginning and end of syntax diagram lines.

Symbol	Indicates
▶▶—	the syntax diagram starts here
—▶	the syntax diagram is continued on the next line
▶—	the syntax diagram is continued from the previous line
—▶▶	the syntax diagram ends here

- Required items appear on the horizontal line (the main path).

▶▶—STATEMENT—*required-item*—▶▶

- Optional items appear below the main path.

▶▶—STATEMENT—optional-item—▶▶

- Keywords appear in uppercase (for example, STATEMENT). They must be spelled exactly as shown. Variables appear in all lowercase letters and in italics (for example, *item*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other symbols are shown, you must enter them as part of the syntax.

- When you can choose from two or more items, the items appear vertically, in a stack. If you **must** choose one of the items, one item of the stack appears on the main path. The default, if any, appears above the main path and is chosen by the compiler if you do not specify another choice. In some cases, the default is affected by the system in which the program is being run or the environmental parameters specified.

Because *choice 1* appears on the horizontal bar, one of the items must be included in the statement. If you don't specify either *choice 1* or *choice 2*, the compiler implements the default for you.



If choosing one of the items is optional, the entire stack appears below the main path.



- An arrow returning to the left above the main line is a *repeat arrow*, and it indicates an item that can be repeated.



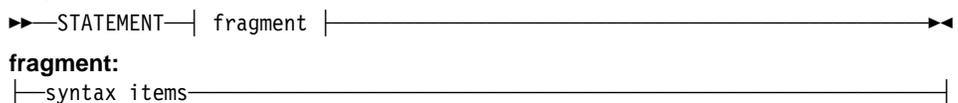
A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

- If there is a comma as part of the repeat arrow, you must use a comma to separate items in a series.



If the comma appears below the repeat arrow line instead of on the line as shown in the previous example, the comma is optional as a separator of items in a series.

- A syntax fragment is delimited in the main syntax diagram by a set of vertical lines. The corresponding meaning of the fragment begins with the name of the fragment followed by the syntax, which starts and ends with a vertical line.



## How to Read the Notational Symbols

Some of the programming syntax in this book is presented using notational symbols. This is to maintain consistency with descriptions of the same syntax in other IBM publications, or to allow the syntax to be shown on single lines within a table or heading.

- **Braces**, { }, indicate a choice of entry. Unless an item is underlined, indicating a default, or the items are enclosed in brackets, you must choose at least one of the entries.

- Items separated by a single **vertical bar**, |, are alternative items. You can select only one of the group of items separated by single vertical bars. (Double vertical bars, ||, specify a concatenation operation, not alternative items. See the *PL/I for MVS & VM Language Reference* for more information on double vertical bars.)
- Anything enclosed in **brackets**, [ ], is optional. If the items are vertically stacked within the brackets, you can specify only one item.
- An **ellipsis**, ..., indicates that multiple entries of the type immediately preceding the ellipsis are allowed.

### Example of Notation

The following example of PL/I syntax illustrates the notational symbols described in “How to Read the Notational Symbols” on page xix:

```
DCL file-reference FILE STREAM
      INPUT | {OUTPUT [PRINT]}
      ENVIRONMENT(option ...);
```

Interpret this example as follows:

- You must spell and enter the first line as shown, except for *file-reference*, for which you must substitute the name of the file you are referencing.
- In the second line, you can specify INPUT or OUTPUT, but not both. If you specify OUTPUT, you can optionally specify PRINT as well. If you do not specify either alternative, INPUT takes effect by default.
- You must enter and spell the last line as shown (including the parentheses and semicolon), except for *option ...*, for which you must substitute one or more options separated from each other by one or more blanks.

---

## Part 2. Compiling your program

<b>Chapter 1. Using Compile-Time Options and Facilities</b> . . . . .	5
Compile-Time Option Descriptions . . . . .	5
AGGREGATE . . . . .	8
ATTRIBUTES . . . . .	8
CMPAT . . . . .	8
COMPILE . . . . .	10
CONTROL . . . . .	10
DECK . . . . .	10
ESD . . . . .	11
FLAG . . . . .	11
GONUMBER . . . . .	11
GOSTMT . . . . .	12
GRAPHIC . . . . .	12
IMPRECISE . . . . .	12
INCLUDE . . . . .	13
INSOURCE . . . . .	13
INTERRUPT . . . . .	13
LANGLVL . . . . .	14
LINECOUNT . . . . .	14
LIST . . . . .	14
LMESSAGE . . . . .	15
MACRO . . . . .	15
MAP . . . . .	15
MARGINI . . . . .	16
MARGINS . . . . .	16
MDECK . . . . .	17
NAME . . . . .	17
NEST . . . . .	18
NOT . . . . .	18
NUMBER . . . . .	18
OBJECT . . . . .	19
OFFSET . . . . .	19
OPTIMIZE . . . . .	20
OPTIONS . . . . .	20
OR . . . . .	21
SEQUENCE . . . . .	21
SIZE . . . . .	22
SMESSAGE . . . . .	23
SOURCE . . . . .	23
STMT . . . . .	23
STORAGE . . . . .	23
SYNTAX . . . . .	24
SYSTEM . . . . .	24
TERMINAL . . . . .	25
TEST . . . . .	26
XREF . . . . .	27
Input Record Formats . . . . .	28
Specifying Options in the %PROCESS or *PROCESS statements . . . . .	28
Using the Preprocessor . . . . .	29
Invoking the Preprocessor . . . . .	29

Using the %INCLUDE Statement . . . . .	30
Using the PL/I Preprocessor in Program Testing . . . . .	31
Using % Statements . . . . .	32
Invoking the Compiler from an Assembler Routine . . . . .	32
Option List . . . . .	33
DDNAME List . . . . .	33
Page Number . . . . .	33
Using the Compiler Listing . . . . .	33
Heading Information . . . . .	34
Options Used for the Compilation . . . . .	34
Preprocessor Input . . . . .	34
SOURCE Program . . . . .	34
Statement Nesting Level . . . . .	36
ATTRIBUTE and Cross-Reference Table . . . . .	36
Attribute Table . . . . .	37
Cross-Reference Table . . . . .	37
Aggregate Length Table . . . . .	38
Storage Requirements . . . . .	39
Statement Offset Addresses . . . . .	40
External Symbol Dictionary . . . . .	41
ESD Entries . . . . .	42
Static Internal Storage Map . . . . .	43
Object Listing . . . . .	44
Messages . . . . .	44
Return Codes . . . . .	45
<b>Chapter 2. Using PL/I Cataloged Procedures under MVS . . . . .</b>	<b>46</b>
IBM-Supplied Cataloged Procedures . . . . .	46
Compile Only (IEL1C) . . . . .	47
Compile and Link-Edit (IEL1CL) . . . . .	48
Compile, Link-Edit, and Run (IEL1CLG) . . . . .	50
Compile, Load and Run (IEL1CG) . . . . .	51
Invoking a Cataloged Procedure . . . . .	52
Specifying Multiple Invocations . . . . .	52
Link-Editing Multitasking Programs . . . . .	53
Modifying the PL/I Cataloged Procedures . . . . .	54
EXEC Statement . . . . .	54
DD Statement . . . . .	55
<b>Chapter 3. Compiling under MVS . . . . .</b>	<b>56</b>
Invoking the Compiler under TSO . . . . .	56
Allocating Data Sets . . . . .	57
Using the PLI Command . . . . .	59
Example 1 . . . . .	61
Example 2 . . . . .	62
Compiler Listings . . . . .	62
Using %INCLUDE under TSO . . . . .	62
Allocating Data Sets in %INCLUDE . . . . .	63
Running Jobs in a Background Region . . . . .	63
Using JCL during Compilation . . . . .	64
EXEC Statement . . . . .	64
DD Statements for the Standard Data Sets . . . . .	64
Input (SYSIN or SYSCIN) . . . . .	65
Output (SYSLIN, SYSPUNCH) . . . . .	65

Temporary Workfile (SYSUT1)	66
Statement Lengths	66
Listing (SYSPRINT)	67
Source Statement Library (SYSLIB)	67
Example of Compiler JCL	67
Specifying Options	68
Specifying Options in the EXEC Statement	68
Compiling Multiple Procedures in a Single Job Step	69
SIZE Option	69
NAME Option	69
Return Codes in Batched Compilation	70
Job Control Language for Batched Processing	70
Examples of Batched Compilations	71
Correcting Compiler-Detected Errors	71
The PL/I Compiler and MVS/ESA	71
Compiling for CICS	72
<b>Chapter 4. Compiling under VM</b>	<b>73</b>
Using the PLIOPT Command	73
Compiler Output and Its Destination	73
Compile-Time Options	74
Files Used by the Compiler	74
PLIOPT Command Options	75
%INCLUDE Statement	75
Example of Using %INCLUDE	76
PLIOPT Command Format	76
Examples:	76
Special Action Will Be Required:	77
PL/I Batched Compilation	78
Correcting Compiler-Detected Errors	78
<b>Chapter 5. Link-Editing and Running</b>	<b>80</b>
Selecting Math Results at Link-Edit Time	80
VM Run-Time Considerations	80
Separately Compiled PL/I MAIN Programs	81
Using Data Sets and Files	81
Using VM Files — Example	82
Using VSAM Data Sets — Example	83
Using OS Data Sets — Example	84
Restrictions Using PL/I under VM	85
Using Record I/O at the Terminal	85
PL/I Conventions under VM	86
Formatting Conventions for PRINT Files	86
Changing the Format on PRINT Files	86
Automatic Prompting	86
Punctuating Long Input Lines	87
Punctuating GET LIST and GET DATA Statements	88
ENDFILE	88
DISPLAY and REPLY under VM	89
MVS Run-Time Considerations	89
Formatting Conventions for PRINT Files	89
Changing the Format on PRINT Files	89
Automatic Prompting	90
Punctuating Long Input Lines	91

Punctuating GET LIST and GET DATA Statements . . . . .	91
ENDFILE . . . . .	92
SYSPRINT Considerations . . . . .	92

---

# Chapter 1. Using Compile-Time Options and Facilities

This chapter describes the options that you can use for the compiler, along with their abbreviations and IBM-supplied defaults. It's important to remember that PL/I requires access to Language Environment run time when you compile your applications.

Most compile-time options have a positive and negative form. The negative form is the positive with 'NO' added at the beginning (as in TEST and NOTEST). Some options have only a positive form (as in SYSTEM).

Your installation can change the IBM-supplied defaults when this product is installed. Therefore, the defaults listed in this chapter might not be the same as those chosen by your installation. You can override most defaults when you compile your PL/I program.

---

## Compile-Time Option Descriptions

There are three types of compile-time options:

1. Simple pairs of keywords: a positive form that requests a facility, and an alternative negative form that inhibits that facility (for example, NEST and NONEST).
2. Keywords that allow you to provide a value list that qualifies the option (for example, FLAG(W)).
3. A combination of 1 and 2 above (for example, NOCOMPILE(E)).

Table 3 lists all compile-time options with their abbreviated syntax and their IBM-supplied default values. Table 4 on page 7 lists the options by function so that you can, for example, determine the preprocessing options.

The paragraphs following Table 3 and Table 4 describe the options in alphabetical order. In the accompanying syntax diagrams, defaults are not highlighted because that information is provided Table 3. For those options that specify that the compiler is to list information, only a brief description is included; the generated listing is described under "Using the Compiler Listing" on page 33.

**Note:** Under VM, use only the abbreviated form of the compile-time option if the option name is longer than eight characters.

---

*Table 3 (Page 1 of 2). Compile-Time Options, Abbreviations, and IBM-Supplied Defaults*

Compile-time option	Abbreviated name	MVS default	TSO default	VM default
AGGREGATE NOAGGREGATE	AG NAG	NAG	NAG	NAG
ATTRIBUTES[(FULL SHORT)] NOATTRIBUTES	A[(F S)] NA	NA [(FULL)] <sup>1</sup>	NA [(FULL)] <sup>1</sup>	NA [(FULL)] <sup>1</sup>
CMPAT(V1 V2)	CMP(V1 V2)	CMP(V2)	CMP(V2)	CMP(V2)
COMPILE NOCOMPILE[(W E S)]	C NC[(W E S)]	NC(S)	NC(S)	NC(S)
CONTROL('password')	-	-	-	-
DECK NODECK	D ND	ND	ND	ND
ESD NOESD	-	NOESD	NOESD	NOESD
FLAG[(I W E S)]	F[(I W E S)]	F(I)	F(W)	F(W)
GONUMBER NOGONUMBER	GN NGN	NGN	NGN	NGN

Table 3 (Page 2 of 2). Compile-Time Options, Abbreviations, and IBM-Supplied Defaults

Compile-time option	Abbreviated name	MVS default	TSO default	VM default
GOSTMT NOGOSTMT	GS NGS	NGS	NGS	NGS
GRAPHIC NOGRAPHIC	GR NGR	NGR	NGR	NGR
IMPRECISE NOIMPRESISE	IMP NIMP	NIMP	NIMP	NIMP
INCLUDE NOINCLUDE	INC NINC	NINC	NINC	NINC
INSOURCE NOINSOURCE	IS NIS	IS	NIS	NIS
INTERRUPT NOINTERRUPT	INT NINT	NINT	NINT	NINT
LANGLVL((OS,SPROG NOSPROG))	-	LANGLVL (OS,NOSPROG)	LANGLVL (OS,NOSPROG)	LANGLVL (OS,NOSPROG)
LINECOUNT(n)	LC(n)	LC(55)	LC(55)	LC(55)
LIST((m[,n])) NOLIST	-	NOLIST	NOLIST	NOLIST
LMESSAGE SMESSAGE	LMSG SMMSG	LMSG	LMSG	LMSG
MACRO NOMACRO	M NM	NM	NM	NM
MAP NOMAP	-	NOMAP	NOMAP	NOMAP
MARGIN('c') NOMARGIN	MI('c') NMI	NMI	NMI	NMI
MARGINS(m,n[,c])	MAR(m,n[,c])	MAR F-format: (2,72) V-format: (10,100)	MAR F-format: (2,72) V-format: (10,100)	MAR F-format: (2,72) V-format: (10,100)
MDECK NOMDECK	MD NMD	NMD	NMD	NMD
NAME('name')	N('name')	-	-	-
NOT	-	NOT(' -')	NOT(' -')	NOT(' -')
NEST NONEST	-	NONEST	NONEST	NONEST
NUMBER NONUMBER	NUM NNUM	NNUM	NUM	NUM
OBJECT NOOBJECT	OBJ NOBJ	OBJ	OBJ	OBJ
OFFSET NOOFFSET	OF NOF	NOF	NOF	NOF
OPTIMIZE(TIME 0 2) NOOPTIMIZE	OPT(TIME 0 2) NOPT	NOPT	NOPT	NOPT
OPTIONS NOOPTIONS	OP NOP	OP	NOP	NOP
OR	-	OR(' ')	OR(' ')	OR(' ')
SEQUENCE(m,n) NOSEQUENCE	SEQ(m,n) NSEQ	SEQ F-format: (73,80) V-format: (1,8)	SEQ F-format: (73,80) V-format: (1,8)	SEQ F-format: (73,80) V-format: (1,8)
SIZE([-jyyyyyyyy [-jyyyyyK MAX])	SZ([-jyyyyyyyy  [-jyyyyyK MAX])	SZ(MAX)	SZ(MAX)	SZ(MAX)
SOURCE NOSOURCE	S NS	S	NS	NS
STMT NOSTMT	-	STMT	NOSTMT	NOSTMT
STORAGE NOSTORAGE	STG NSTG	NSTG	NSTG	NSTG
SYNTAX NOSYNTAX[(W E S)]	SYN NSYN[(W E S)]	NSYN(S)	NSYN(S)	NSYN(S)
SYSTEM(CMS CMSTPL MVS TSO CICS IMS)	-	MVS	MVS	VM
TERMINAL[(opt-list)] NOTERMINAL	TERM[(opt-list)] NTERM	NTERM	TERM	TERM
TEST([(ALL BLOCK NONE PATH STMT) [.SYM .NOSYM]]) NOTEST	-	NOTEST [(NONE,SYM)] <sup>2</sup>	NOTEST [(NONE,SYM)] <sup>2</sup>	NOTEST [(NONE,SYM)] <sup>2</sup>
XREF[(FULL SHORT)] NOXREF	X[(F S)] NX	NX [(FULL)] <sup>1</sup>	NX [(FULL)] <sup>1</sup>	NX [(FULL)] <sup>1</sup>

**Notes:**

1. FULL is the default suboption if the suboption is omitted with ATTRIBUTES or XREF.
2. (NONE,SYM) is the default suboption if the suboption is omitted with TEST.

Table 4 (Page 1 of 2). Compile-Time Options Arranged by Function

<b>Options for use when testing or debugging</b>	<b>TEST</b>	specifies which debugging tool capabilities are available for testing programs.
<b>Listing options</b>		
<i>Control listings produced</i>	<b>AGGREGATE</b>	lists aggregates and their size.
	<b>ATTRIBUTES</b>	lists attributes of identifiers.
	<b>ESD</b>	lists external symbol dictionary.
	<b>FLAG</b>	suppresses diagnostic messages below a certain severity.
	<b>INSOURCE</b>	lists preprocessor input.
	<b>LIST</b>	lists object code produced by compiler.
	<b>MAP</b>	lists offsets of variables in static control section and DSAs.
	<b>OFFSET</b>	lists statement numbers associated with offsets.
	<b>OPTIONS</b>	lists options used.
	<b>SOURCE</b>	lists source program or preprocessor output.
	<b>STORAGE</b>	lists storage used.
	<b>XREF</b>	lists statements in which each identifier is used.
<i>Improve readability of source listing</i>	<b>MARGINI</b>	highlights any source outside margins.
	<b>NEST</b>	indicates do-group and block level by numbering in margin.
<i>Control lines per page</i>	<b>LINECOUNT</b>	specifies number of lines per page on listing.
<b>Input options</b>	<b>GRAPHIC</b>	specifies that shift codes can be used in source.
	<b>MARGINS</b>	identifies position of PL/I source and a carriage control character.
	<b>NOT</b>	used to specify up to seven alternate symbols for the logical NOT operator.
	<b>OR</b>	used to specify up to seven alternate symbols for the logical OR operator and the string concatenation operator.
	<b>SEQUENCE</b>	specifies the columns used for sequence numbers.
<b>Options to prevent unnecessary processing</b>	<b>COMPILE</b>	stops processing after errors are found in syntax checking.
	<b>SYNTAX</b>	stops processing after errors are found in preprocessing.
<b>Options for preprocessing</b>	<b>INCLUDE</b>	allows secondary input to be included without using preprocessor.
	<b>MACRO</b>	allows preprocessor to be used.
	<b>MDECK</b>	produces a source deck from preprocessor output.
<b>Option to improve performance</b>	<b>OPTIMIZE</b>	improves run-time performance or specifies faster compile time.
<b>Options to use when producing an object module</b>	<b>CMPAT</b>	controls level of compatibility with previous releases.
	<b>DECK</b>	produces an object module in punched card format.
	<b>OBJECT</b>	produces object code.
	<b>NAME</b>	specifies the TEXT file will be given a particular external name.
	<b>SYSTEM</b>	specifies the parameter list format that is passed to the main procedure.
<b>Option to control storage</b>	<b>SIZE</b>	controls the amount of storage used by the compiler.
<b>Options to improve usability at a terminal</b>	<b>LMESSAGE/SMESSAGE</b>	specifies full or concise message format.
	<b>TERMINAL</b>	specifies how much of listing is transmitted to terminal.
<b>Options to specify statement numbering system</b>	<b>NUMBER &amp; GONUMBER</b>	numbers statements according to line in which they start.
	<b>STMT &amp; GOSTMT</b>	numbers statements sequentially.
<b>Option to control effect of attention interrupts</b>	<b>INTERRUPT</b>	specifies that the ATTENTION condition will be raised after an interrupt is caused.
<b>Option for use on imprecise interrupt machines</b>	<b>IMPRECISE</b>	allows imprecise interrupts to be handled correctly.
<b>Option to control compile-time options</b>	<b>CONTROL</b>	specifies that any compile-time options previously deleted are available.

Table 4 (Page 2 of 2). Compile-Time Options Arranged by Function

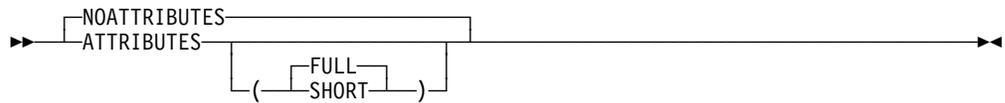
<b>Option to control language level</b>	<b>LANGLVL</b>	defines the level of language supported.
---	----------------	--

## AGGREGATE



The AGGREGATE option specifies that the compiler includes an aggregate-length table that gives the lengths of all arrays and major structures in the source program in the compiler listing.

## ATTRIBUTES



The ATTRIBUTES option specifies that the compiler includes a table of source-program identifiers and their attributes in the compiler listing. If you include both ATTRIBUTES and XREF, the two tables are combined. However, if the SHORT and FULL suboptions are in conflict, the last option specified is used. For example, if you specify ATTRIBUTES(SHORT) XREF(FULL), FULL applies to the combined listing.

### FULL

All identifiers and attributes are included in the compiler listing. FULL is the default.

### SHORT

Unreferenced identifiers are omitted, making the listing more manageable.

## CMPAT



The CMPAT option specifies whether object compatibility with OS PL/I Version 1 is maintained for those programs sharing arrays, AREAs, or aggregates.

### CMPAT(V1)

If you use CMPAT(V1), you can use arrays, AREAs, and aggregates in exactly the same way that they were used in OS PL/I Version 1 as long as other external procedures sharing them are not compiled with CMPAT(V2).

If any procedures in an application load module (MAIN or FETCHed) are recompiled (and therefore relink-edited), object code compatibility with OS PL/I Version 1 Release 5.1 is provided under the following guidelines:

- If arrays, aggregates, or AREAs are to be shared between OS PL/I Version 1 Release 5.1 object code and PL/I MVS & VM object code, PL/I MVS & VM compilations must use CMPAT(V1).
- If arrays, aggregates, or AREAs are to be shared between PL/I MVS & VM object code only, PL/I MVS & VM compilations must use either CMPAT(V1) or CMPAT(V2), but not both.
- Using CMPAT(V2) is required for larger arrays, aggregates, or AREAs and is recommended even if you do not use larger arrays, aggregates, or AREAs.
- If arrays, aggregates, or AREAs are to be shared between OS PL/I Version 1 Release 5.1 object code only, no precautions need to be taken.

### **CMPAT(V2)**

In general, you should compile PL/I programs with CMPAT(V2).

CMPAT(V2) does not provide object compatibility with OS PL/I Version 1. Therefore, if you are migrating OS PL/I Version 1 applications or OS PL/I Version 2 applications compiled with CMPAT(V1), you must make code changes if:

- You want to use fullword subscripts.
- You have any expressions that rely on precision and scale values returned from the BUILTIN functions HBOUND, LBOUND, DIM, or ALLOCATION.

If you do not have either of the above requirements you do not need to make code changes to use CMPAT(V2) as long as all external procedures sharing the same array or aggregate are also compiled with CMPAT(V2).

If all of your existing object code was produced by OS PL/I Version 2 with the compiler option CMPAT(V2), your object code is fully compatible with object code produced by PL/I MVS & VM, provided you continue to use CMPAT(V2) compiler option. (Other factors can affect object code compatibility. For a list of these factors, see *PL/I for MVS & VM Compiler and Run-Time Migration Guide*.)

If some or all of your existing object code was produced by OS PL/I Version 2 with the compiler option CMPAT(V1) or by OS PL/I Version 1 Release 5.1, the following considerations apply when mixing with object code produced by PL/I MVS & VM:

- If arrays, aggregates, or AREAs are to be shared between OS PL/I Version 1 Release 5.1 or OS PL/I Version 2 (compiled with CMPAT(V1)) object code and PL/I MVS & VM object code, PL/I MVS & VM compilations must use CMPAT(V1).
- If arrays, aggregates, or AREAs are to be shared between OS PL/I Version 2 (compiled with CMPAT(V2)) object code and PL/I MVS & VM object code, PL/I MVS & VM compilations must use CMPAT(V2).

Using CMPAT(V2) is required for larger arrays, aggregates, or AREAs and is recommended even if you do not use larger arrays, aggregates, or AREAs.

## COMPILE



The COMPILE option specifies that the compiler compiles the source program unless it detects an unrecoverable error during preprocessing or syntax checking. Whether the compiler continues or not depends on the severity of the error detected, as specified by the NOCOMPILER option in the list below. The NOCOMPILER option specifies that processing stops unconditionally after syntax checking.

### **NOCOMPILER(W)**

No compilation if a warning, error, severe error, or unrecoverable error is detected.

### **NOCOMPILER(E)**

No compilation if an error, severe error, or unrecoverable error is detected.

### **NOCOMPILER(S)**

No compilation if a severe error or unrecoverable error is detected.

If the compilation is terminated by the NOCOMPILER option, the cross-reference listing and attribute listing can be produced; the other listings that follow the source program will not be produced.

## CONTROL



The CONTROL option specifies that any compile-time options deleted for your installation are available for this compilation. Using the CONTROL option alone does not restore compile-time options you have deleted from your system. You still must specify the appropriate keywords to use the options. The CONTROL option must be specified with a password that is established for each installation. If you use an incorrect password, processing will be terminated. If you use the CONTROL option, it must be specified first in the list of options.

### **password**

is a character string not exceeding eight characters.

**Under VM:** You cannot use a right or left parenthesis or include lower case characters on a password if you use CONTROL in the PLIOPT command.

## DECK



The DECK option specifies that the compiler produces an object module in the form of 80-character records and store it in the SYSPUNCH data set. Columns 73-76 of each record contain a code to identify the object module. This code comprises the



## GOSTMT



The GOSTMT option specifies that the compiler produces additional information that allows statement numbers from the source program to be included in run-time messages.

These statement numbers can also be derived by using the offset address, which is always included in run-time messages, and the table produced by the OFFSET option. (The STMT option must also apply.)

The GOSTMT option implies STMT, NONUMBER, and NOGONUMBER. If STMT applies, GOSTMT is forced by the ALL, STMT, and PATH suboptions of the TEST option. The OFFSET option is separate from these numbering options and must be specified if required.

## GRAPHIC



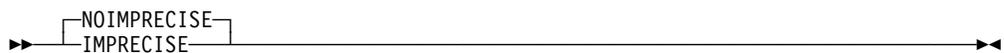
Using GRAPHIC option specifies that the source program can contain double-byte characters. The hexadecimal codes 'OE' and 'OF' are treated as the shift-out and shift-in control codes, respectively, wherever they appear in the source program. This includes occurrences in comments and string constants.

The GRAPHIC compile-time option must be specified if the source program uses any of the following:

- DBCS identifiers
- Graphic string constants
- Mixed string constants
- Shift codes anywhere else in the source

For more information see the discussion of the DBCSOS Ordering Product and the SIZE option on page 22.

## IMPRECISE



The IMPRECISE option specifies that the compiler includes extra text in the object module to localize imprecise interrupts when executing the program with an IBM System/390 Model 165 or 195. This extra text is generated for ON statements (to ensure that the correct ON-units are entered if interrupts occur), for null statements, and for ENTRY statements. The correct line or statement numbers do not necessarily appear in run-time messages. If you need more accurate identification of the statement in error, insert null statements at suitable points in your program.

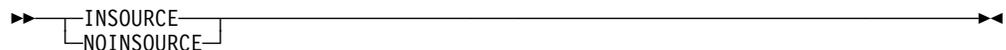
## INCLUDE



The INCLUDE option specifies that %INCLUDE statements are handled without using the full preprocessor facilities and incurring more overhead. This method is faster than using the preprocessor for programs that use the %INCLUDE statement but no other PL/I preprocessor statements. The INCLUDE option has no effect if preprocessor statements other than %INCLUDE are used in the program. In these cases, the MACRO option must be used.

If you specify the MACRO option, it overrides the INCLUDE option.

## INSOURCE



The INSOURCE option specifies that the compiler should include a listing of the source program before the PL/I macro preprocessor translates it. Thus, the INSOURCE listing contains preprocessor statements that do not appear in the SOURCE listing. This option is applicable only when the MACRO option is in effect.

## INTERRUPT



This option determines the effect of attention interrupts when the compiled PL/I program runs under an interactive system. (If specified on a batch system, INTERRUPT can cause an abend.)

The INTERRUPT option causes the compiled program to respond to attention requests (interrupts). If you have written a program that relies on raising the ATTENTION condition, you must compile it with the INTERRUPT option.

The INTERRUPT option allows attention interrupts to become an integral part of programming. This gives you considerable interactive control of the program.

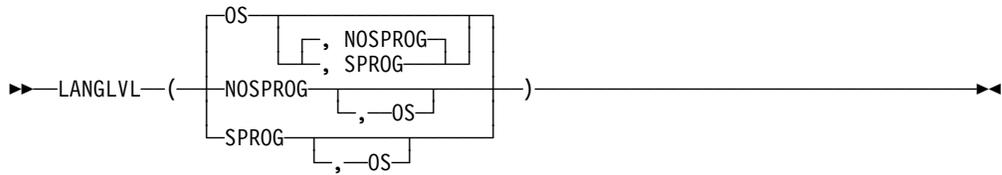
If you specify the INTERRUPT option, an established ATTENTION ON-unit gets control when an attention interrupt occurs. When the execution of an ATTENTION ON-unit is complete, control returns to the point of interrupt unless directed elsewhere by means of a GOTO statement. If you do not establish an ATTENTION ON-unit, the attention interrupt is ignored.

If you specify NOINTERRUPT, an attention interrupt during a program run does not give control to any ATTENTION ON-units.

If you require the attention interrupt capability purely for testing purposes, you need not use the INTERRUPT option. The TEST option provides this capability. For more information See "TEST" on page 26.

See Chapter 20, “Interrupts and Attention Processing” on page 436 for more information about the INTERRUPT option.

## LANGLVL



The LANGLVL option specifies the level of PL/I language supported, including whether pointers in expressions are to be supported.

### OS

specifies the level of PL/I language the compiler is to support. OS is the only level currently supported.

### NOSPROG

specifies that the compiler is *not* to allow the additional support for pointers allowed under SPROG.

### SPROG

specifies that the compiler is to allow extended operations on pointers, including arithmetic, and the use of the POINTERADD, BINARYVALUE, and POINTERVERVALUE built-in functions.

For more information on pointer operations, see the *PL/I for MVS & VM Language Reference* book.

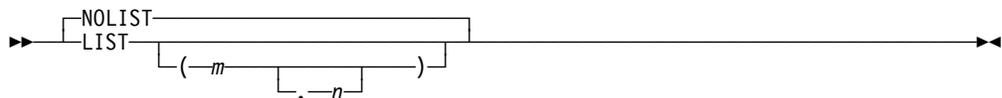
## LINECOUNT



The LINECOUNT option specifies the number of lines included in each page of the compiler listing, including heading lines and blank lines.

**n** is the number of lines. It must be in the range 1 through 32,767, but only headings are generated if you specify less than 7. When you specify less than 100, the static internal storage map and the object listing are printed in double column format. Otherwise, they are printed in single column format.

## LIST



The LIST option specifies that the compiler includes a listing of the object module (in a syntax similar to assembler language instructions) in the compiler listing. If both *m* and *n* are omitted, the compiler produces a listing of the whole program.

**m** is the number of the first, or only, source statement for which an object listing is required.

**n** is the number of the last source statement for which an object listing is required. If *n* is omitted, only statement *m* is listed.

If the option NUMBER applies, *m* and *n* must be specified as line numbers. If the STMT option applies, *m* and *n* must be statement numbers.

If you use LIST in conjunction with MAP, it increases the information generated by MAP. (See “MAP” for more information on the MAP compile-time option.)

**Under TSO:** Use the LIST(*m*,*n*) option to direct a listing of particular statements to the terminal in either of the following ways:

- Use the LIST option, with no statement numbers, within the TERMINAL option.
- Use the PRINT(\*) operand in the PLI command.

## LMESSAGE



The LMESSAGE and SMESSAGE options produce messages in a long form (specify LMESSAGE) or in a short form (specify SMESSAGE).

## MACRO



The MACRO option specifies that the source program is to be processed by the preprocessor. MACRO overrides INCLUDE if both are specified.

## MAP



The MAP option specifies that the compiler produces tables showing the organization of the static storage for the object module. These tables show how variables are mapped in the static internal control section and in DSAs, thus enabling STATIC INTERNAL and AUTOMATIC variables to be found in PLIDUMP. If LIST (described under “LIST” on page 14) is also specified, the MAP option produces tables showing constants, control blocks and INITIAL variable values. LIST generates a listing of the object code in pseudo-assembler language format.

If you want a complete map, but not a complete list, you can specify a single statement as an argument for LIST to minimize the size of the LIST. For example:

```
%PROCESS MAP LIST(1);
```

## MARGINI

► `NOMARGINI`  
`MARGINI` (`'-c'`) ►

The MARGINI option specifies that the compiler includes a specified character in the column preceding the left-hand margin, and also in the column following the right-hand margin of the listings that the compiler produces when you use the INSOURCE and SOURCE options. The compiler shifts any text in the source input that precedes the left-hand margin left one column. It shifts any text that follows the right-hand margin right one column. Thus you can easily detect text outside the source margins.

**c** is the character to be printed as the margin indicator.

## MARGINS

► `MARGINS` (`-m, -n` `[, -c]`) ►

The MARGINS option specifies which part of each compiler input record contains PL/I statements, and the position of the ANS control character that formats the listing, if the SOURCE and/or INSOURCE options apply. The compiler does not process data that is outside these limits, but it does include it in the source listings.

The PL/I source is extracted from the source input records so that the first data byte of a record immediately follows the last data byte of the previous record. For variable records, you must ensure that when you need a blank you explicitly insert it between margins of the records.

**m** is the column number of the leftmost character (first data byte) that is processed by the compiler. It must not exceed 100.

**n** is the column number of the rightmost character (last data byte) that is processed by the compiler. It should be greater than *m*, but not greater than 100.

For variable-length records, *n* is interpreted as the rightmost column, or the last data byte if the record has less than *n* data bytes. Thus, the last character of a variable-length record is usually a nonblank character and is immediately followed (without any intervening blank) by the first data byte (*m*) of the next record. If you do not intend to have continuation, be sure that at least one blank occurs at the beginning (*m*) of the next record.

**c** is the column number of the ANS printer control character. It must not exceed 100 and should be outside the values specified for *m* and *n*. A value of 0 for *c* indicates that no ANS control character is present. Only the following control characters can be used:

(blank)	Skip one line before printing
0	Skip two lines before printing
-	Skip three lines before printing
+	No skip before printing
1	Start new page

Any other character is an error and is replaced by a blank.

Do not use a value of *c* that is greater than the maximum length of a source record, because this causes the format of the listing to be unpredictable. To avoid this problem, put the carriage control character to the left of the source margins for variable length records.

Specifying MARGINS(.,c) is an alternative to using %PAGE and %SKIP statements (described in the *PL/I for MVS & VM Language Reference*).

The IBM-supplied default for fixed-length records is MARGINS(2,72). For variable-length and undefined-length records, the IBM-supplied default is MARGINS(10,100). This specifies that there is **no** printer control character.

Use the MARGINS option to override the default for the primary input in a program. The secondary input must have either the same margins as the primary input if it is the same type of record, or default margins if it is a different type. (See “Input Record Formats” on page 28.)

## MDECK

```
▶ NOMDECK  
MDECK ▶
```

The MDECK option specifies that the preprocessor produces a copy of its output on the file defined by the SYSPUNCH DD statement. The MDECK option allows you to retain the output from the preprocessor as a file of 80-column records.

The compiler ignores MDECK if NOMACRO is in effect.

## NAME

```
▶ NAME (—'—name—'—) ▶
```

The NAME option specifies that the TEXT file created by the compiler is given the specified external name that you specify. This allows you to create more than one TEXT file while doing batched compilation. It also allows you to produce text files that can be included in a text library. You can also use the NAME option to cause the linkage editor to substitute a new load module for an existing load module with the same name in the library.

### **name**

has from one through eight characters, and begins with an alphabetic character. NAME has no default.

For more uses of the NAME option, see either “Compiling a Program to be Placed in a TXTLIB” on page 77 for compiling under VM, or “NAME Option” on page 69 for compiling under MVS.

## NEST



You can use the NEST option to specify that the listing resulting from the SOURCE option indicates the block level and the do-group level for each statement.

## NOT



The NOT option specifies up to seven alternate symbols, any one of which can be used as the logical NOT operator.

### char

is a single SBCS character.

You cannot specify any of the alphabetic characters, digits, and special characters defined in the *PL/I for MVS & VM Language Reference*, except for the logical NOT symbol (¬).

If you specify the NOT option, the standard NOT symbol is no longer recognized unless you specify it as one of the characters in the character string.

For example, NOT('~') means that the tilde character, X'A1', will be recognized as the logical NOT operator, and the standard NOT symbol, '¬', X'5F', will not be recognized. Similarly, NOT('~¬') means that either the tilde or the standard NOT symbol will be recognized as the logical NOT operator.

The IBM-supplied default code point for the NOT symbol is X'5F'. The logical NOT sign might appear as a logical NOT symbol (¬) or a caret symbol (^) on your keyboard.

## NUMBER



The NUMBER option specifies that numbers in the sequence fields in the source input records are used to derive the statement numbers in the listings resulting from the AGGREGATE, ATTRIBUTES, LIST, OFFSET, SOURCE and XREF options.

You can specify the position of the sequence field in the SEQUENCE option.

Otherwise the following default positions are assumed:

- First eight columns for undefined-length or variable-length source input records
- Last eight columns for fixed-length source input records

**Note:** The preprocessor output has fixed-length records regardless of the format of the primary input. The sequence numbers are in columns 73-80 in the source listing.

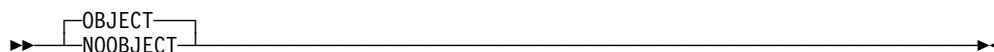
The compiler calculates the line number from the five right-hand characters of the sequence number (or the number specified, if less than five). These characters are converted to decimal digits if necessary. Each time the compiler finds a line number that is not greater than the preceding line number, it forms a new line number by adding the minimum integral multiple of 100,000 to produce a line number that is greater than the preceding one. The compiler issues a message to warn you of the adjustment, except when you specify the INCLUDE option or the MACRO option.

If there is more than one statement on a line, the compiler uses a suffix to identify the actual statement in the messages. For example, the second statement beginning on the line numbered 40 is identified by the number 40.2. The maximum value for this suffix is 31. Thus the thirty-first and subsequent statements on a line have the same number.

If the sequence field consists only of blanks, the compiler forms the new line number by adding 10 to the preceding one. The maximum line number allowed by the compiler is 134,000,000. Numbers that would normally exceed this are set to this maximum value. Only eight digits print in the source listing; line numbers of 100,000,000 or over print without the leading 1 digit.

If you specify NONUMBER, STMT and NOGONUMBER are implied. NUMBER is implied by NOSTMT or GONUMBER.

## OBJECT



The OBJECT option specifies that the compiler creates an object module and stores it in a TEXT file (VM) or in a data set defined by the DD statement with the name SYSLIN (MVS).

## OFFSET

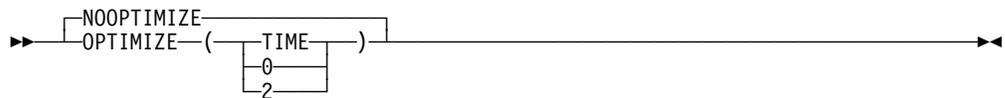


The OFFSET option specifies that the compiler prints a table of statement or line numbers for each procedure with their offset addresses relative to the primary entry point of the procedure. You can use this table to identify a statement from a run-time error message if the GONUMBER or GOSTMT option is not in effect.

If GOSTMT applies, the run-time library includes statement numbers, as well as offset addresses, in run-time messages. If GONUMBER applies, the run-time library includes line numbers, as well as offset addresses, in run-time messages.

For more information on determining line numbers from the offsets given in error messages, see "Statement Offset Addresses" on page 40.

## OPTIMIZE



The OPTIMIZE option specifies the type of optimization required:

### **OPTIMIZE(TIME)**

specifies that the compiler optimizes the machine instructions generated to produce a more efficient object program. This type of optimization can also reduce the amount of main storage required for the object module. The use of OPTIMIZE(TIME) could result in a substantial increase in compile time over NOOPTIMIZE. During optimization the compiler can move code to increase run-time efficiency. As a result, statement numbers in the program listing cannot correspond to the statement numbers used in run-time messages.

### **OPTIMIZE(0)**

is the equivalent of NOOPTIMIZE.

### **OPTIMIZE(2)**

is the equivalent of OPTIMIZE(TIME).

### **NOOPTIMIZE**

specifies fast compilation speed, but inhibits optimization.

For a full discussion of optimization, see Chapter 14, “Efficient Programming” on page 305.

## OPTIONS



The OPTIONS option specifies that the compiler includes a list showing the compile-time options to be used during this compilation in the compiler listing. This list includes all options applied by default, those specified in the PARM parameter of an EXEC statement or in the invoking command (PLI or PLIOPT), and those specified in a %PROCESS statement.

**Under TSO:** If the PRINT(\*) operand of the PL/I command applies, the list of options prints at the terminal. This can show the negative forms of the options that cause listings to be produced, even where the positive forms apply. The positive form is shown within the TERMINAL option. This is because the PRINT(\*) operand is implemented by generating a TERMINAL option containing a list of options corresponding to those listings that are printed at the terminal. Specifying the TERMINAL option after the PRINT(\*) operand overrides the TERMINAL option generated by the PRINT(\*) operand.

## OR

► OR—(—'—<sup>char</sup>—'—) —————►

The OR option specifies up to seven alternate symbols, any one of which is interpreted as the logical OR operator (|). These symbols are also used as the concatenation operator, which is defined as two consecutive logical OR symbols.

### char

is a single SBCS character.

You cannot specify any of the alphabetic characters, digits, and special characters defined in the *PL/I for MVS & VM Language Reference*, except for the logical OR symbol (|).

If you specify the OR option, the standard OR symbol is no longer recognized unless you specify it as one of the characters in the character string.

For example, OR('\') means that the backslash character, X'E0', will be recognized as the logical OR operator, and two consecutive backslashes will be recognized as the concatenation operator. The standard OR symbol, '|', X'4F', will not be recognized as either operator. Similarly, OR('\|') means that either the backslash or the standard OR symbol will be recognized as the logical OR operator, and either symbol or both symbols can be used to form the concatenation operator.

The IBM-supplied default code point for the OR symbol (|) is X'4F'.

## SEQUENCE

► <sup>SEQUENCE</sup>—(—*m*—, —*n*—) <sub>NOSEQUENCE</sub> —————►

The SEQUENCE option defines the section of the input record from which the compiler takes the sequence numbers. These numbers are included in the source listings produced by the INSOURCE and SOURCE option.

The compiler uses sequence numbers to calculate statement numbers if the NUMBER option is in effect. The compiler does not sort the input lines or records into the specified sequence.

**m** specifies the column number of the left-hand margin.

**n** specifies the column number of the right-hand margin.

The extent specified should not overlap with the source program (as specified in the MARGINS option).

The IBM-supplied default for fixed-length records is SEQUENCE (73,80); for variable-length and undefined-length records. The default is SEQUENCE (1,8).

If the SEQUENCE option is used, an external procedure cannot contain more than 32,767 lines. To compile an external procedure containing more than 32,767 lines, you must specify the NOSEQUENCE option. Because NUMBER and GONUMBER imply SEQUENCE, you should not specify the SEQUENCE or NOSEQUENCE options.

You can use the SEQUENCE option to override the default margin positions that are set up during compiler installation by the FSEQUENCE and VSEQUENCE options (see “Input Record Formats” on page 28).

The FSEQUENCE default applies to F-format records and the VSEQUENCE default applies to V-format or U-format records. Only one of these defaults is overridden by the SEQUENCE option. If the first input record to the compiler is F-format, the FSEQUENCE default is overridden. If the first input record is a V-format or a U-format record, the VSEQUENCE default is overridden. The compiler assumes default values if it encounters a record with a different type of format. The compiler includes numbers that it finds in the sequence field in the source listings produced by the FORMAT, INSOURCE, and SOURCE options.

**Under VM:** Note: The preprocessor output has F-format records regardless of the format of the primary input. The sequence numbers are in columns 73-80 in the source listing.

## SIZE



You can use this option to limit the amount of main storage the compiler uses. This is of value, for example, when dynamically invoking the compiler, to ensure that space is left for other purposes. There are five forms of the SIZE option:

### SIZE(yyyyyyy)

specifies that yyyyyyy bytes of main storage are requested. Leading zeros are not required.

### SIZE(yyyyyK)

specifies that yyyyyK bytes of main storage are requested (1K=1024). Leading zeros are not required.

### SIZE(MAX)

specifies that the compiler obtains as much main storage as it can.

### SIZE(-yyyyyy)

specifies that the compiler obtains as much main storage as it can, and then releases yyyyy bytes to the operating system. Leading zeros are not required.

### SIZE(-yyyK)

specifies that the compiler obtains as much main storage as it can, and then releases yyyK bytes to the operating system (1K=1024). Leading zeros are not required.

The IBM-supplied default, SIZE(MAX), allows the compiler to use as much main storage in the region as it can.

The negative forms of SIZE can be useful when a certain amount of space must be left free and the maximum size is unknown, or can vary because the job is run in regions of different sizes.

**Under MVS:** If you use the DBCSOS Ordering Product under MVS (a utility to sort DBCS characters), you must reserve storage for the operating system to load it. Specify SIZE(-n) to reserve sufficient storage, where *n* is at least 128K. See “ATTRIBUTE and Cross-Reference Table” on page 36.

**Note:** Specifying both a region size that gives the job or job step all the available storage below the line and the compile-time option SIZE(MAX) can cause storage problems.

**Under TSO:** 10K to 30K bytes of storage must be reserved for the operating system to load TSO routines. The exact amount of storage required depends on which routines are in the link pack area. Specify SIZE(-n) to reserve sufficient storage space, where *n* is at least 10K bytes. For TSO edit mode, *n* must be at least 30K bytes.

**Under VM:** You should always use SIZE(MAX) in VM unless it is essential to limit the space used. If you set a limit in the SIZE option, the value used exceeds that which is specified. That is because storage is handled by a VM/compiler interface routine and not directly by the compiler.

## SMESSAGE

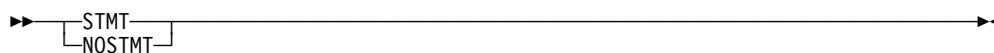
The LMESSAGE and SMESSAGE options produce messages in a long form (specify LMESSAGE) or in a short form (specify SMESSAGE). See “LMESSAGE” on page 15 for the syntax.

## SOURCE



The SOURCE option specifies that the compiler includes a listing of the source program in the compiler listing. The source program listed is either the original source input or, if the MACRO option applies, the output from the preprocessor.

## STMT



The STMT option specifies that statements in the source program are counted, and this statement number is used to identify statements in the compiler listings resulting from the AGGREGATE, ATTRIBUTES, LIST, OFFSET, SOURCE, and XREF options. STMT is implied by NONUMBER or GOSTMT. If NOSTMT is specified, NUMBER and NOGOSTMT are implied.

## STORAGE



The STORAGE option specifies that the compiler includes a table giving the main storage requirements for the object module in the compiler listing.

## SYNTAX



The SYNTAX option specifies that the compiler continues into syntax checking after preprocessing when you specify the MACRO option, unless an unrecoverable error has occurred. Whether the compiler continues with the compilation depends on the severity of the error, as specified by the NOSYNTAX option.

### **NOSYNTAX**

Processing stops unconditionally after preprocessing.

### **NOSYNTAX(W)**

No syntax checking if a warning, error, severe error, or unrecoverable error is detected.

### **NOSYNTAX(E)**

No syntax checking if the compiler detects an error, severe error, or unrecoverable error.

### **NOSYNTAX(S)**

No syntax checking if the compiler detects a severe error or unrecoverable error.

If the SOURCE option applies, the compiler generates a source listing even if it does not perform syntax checking.

If the NOSYNTAX option terminates the compilation, the compiler does not produce the cross-reference listing, attribute listing, and other listings that follow the source program.

You can use this option to prevent wasted runs when debugging a PL/I program that uses the preprocessor.

## SYSTEM



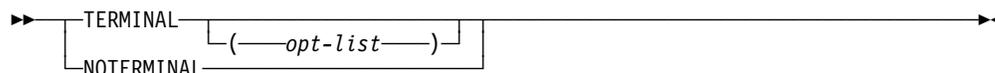
The SYSTEM option specifies the format used to pass parameters to the MAIN PL/I procedure, and generally indicates the host system under which the program runs. MVS, CMS, CMSTPL, CICS, IMS, and TSO are the subparameters recognized. This option allows a program compiled under one system to run under another. For example, a program compiled under VM can run under MVS, and parameters are passed according to MVS conventions.

Table 5 on page 25 shows the type of parameter list you can expect, and how the program runs under the specified host system. It also shows the implied settings of NOEXECOPS.

Table 5. SYSTEM Option Table

SYSTEM option	Type of parameter list	Program runs as	NOEXECOPS implied	For more information
SYSTEM(MVS)	Single varying character string or no parameters.	MVS application program	NO	See <i>Language Environment for MVS &amp; VM Programming Guide</i> .
	Otherwise, arbitrary parameter list.		YES	
SYSTEM(CMS)	Single varying character string or no parameters.	VM application program	NO	See <i>Language Environment for MVS &amp; VM Programming Guide</i> .
	Otherwise, arbitrary parameter list.		YES	
SYSTEM(CMSTPL)	Single varying character string or no parameters.	VM application program	NO	See <i>Language Environment for MVS &amp; VM Programming Guide</i> .
SYSTEM(CICS)	Pointer(s)	CICS transaction	YES	See <i>Language Environment for MVS &amp; VM Programming Guide</i> .
SYSTEM(IMS)	Pointer(s)	IMS application program	YES	See <i>Language Environment for MVS &amp; VM Programming Guide</i> .
SYSTEM(TSO)	Pointer to CCPL	TSO command processor	YES	See <i>Language Environment for MVS &amp; VM Programming Guide</i> .

## TERMINAL



The TERMINAL option is applicable only in a conversational environment. It specifies that a subset of, or all of, the compiler listing produced during compilation prints at the terminal. If you specify TERMINAL without an argument, the compiler prints diagnostic and information messages at the terminal. You can add an argument, which takes the form of an option list, to specify other parts of the compiler listing that the compiler prints at the terminal.

The listing at the terminal is independent of that written on SYSPRINT for TSO, or the LISTING file for VM. However, if you associate SYSPRINT in TSO, or LISTING in VM, with the terminal, only one copy of each option requested is printed.

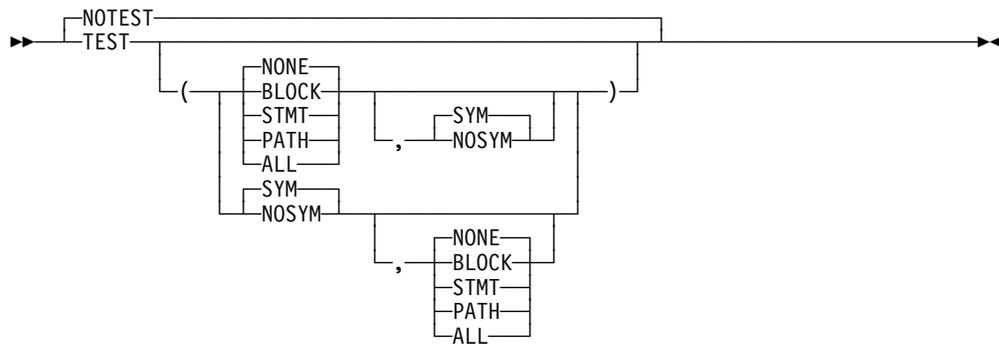
### opt-list

You can specify the following option keywords, their negative forms, or their abbreviated forms, in the option list:

AGGREGATE	OFFSET
ATTRIBUTES	OPTIONS
ESD	SOURCE
INSOURCE	STORAGE
LIST	XREF
MAP	

The other options that relate to the listing (FLAG, GONUMBER, GOSTMT, LINECOUNT, LMESSAGE/SMESSAGE, MARGINI, NEST, NUMBER, and the SHORT and FULL suboptions of ATTRIBUTES and XREF) are the same as for the SYSPRINT listing.

# TEST



The TEST option specifies the level of testing capability that the compiler generates as part of the object code. It allows you to control the location of test hooks and to control whether or not the symbol table will be generated.

The TEST option can imply GONUMBER or GOSTMT, depending on whether NUMBER or STMT is in effect.

Because the TEST option can increase the size of the object code and can affect performance, you might want to limit the number and placement of hooks.

## BLOCK

tells the compiler to insert hooks at block boundaries (block entry and block exit).

## STMT

Specifies that the compiler inserts hooks at statement boundaries and block boundaries. STMT causes a statement table to be generated.

## PATH

tells the compiler to insert hooks:

- Before the first statement enclosed by an iterative DO statement
- Before the first statement of the true part of an IF statement
- Before the first statement of the false part of an IF statement
- Before the first statement of a true WHEN or OTHERWISE statement of a SELECT group
- Before the statement following a user label
- At CALLs or function references—both before and after control is passed to the routine
- At block boundaries.

When PATH is specified, the compiler generates a statement table.

## ALL

tells the compiler to insert hooks at all possible locations and to generate a statement table.

## NONE

tells the compiler not to put hooks into the program.

## **SYM**

tells the compiler to create a symbol table that will allow you to examine variables by name.

## **NOSYM**

tells the compiler not to generate a symbol table.

## **NOTEST**

suppresses the generation of all testing information.

Any TEST option other than NOTEST and TEST(NONE,NOSYM) will automatically provide the attention interrupt capability for program testing.

If the program has an ATTENTION ON-unit that you want invoked, you must compile the program with either of the following:

- The INTERRUPT option
- A TEST option other than NOTEST or TEST(NONE,NOSYM).

## **XREF**



The XREF option specifies that the compiler includes a cross-reference table of names used in the program together with the numbers of the statements in which they are declared or referenced in the compiler listing. (The only exception is that label references on END statements are not included. For example, assume that statement number 20 in the procedure PROC1 is END PROC1;. In this situation, statement number 20 does not appear in the cross reference listing for PROC1.)

## **FULL**

is the default suboption. All identifiers and attributes are included in the compiler listing.

## **SHORT**

Unreferenced identifiers are omitted from the compiler listing.

For a description of the format and content of the cross-reference table, see “Cross-Reference Table” on page 37.

For more information about sorting identifiers and storage requirements with DBCS Ordering Support Product, see “ATTRIBUTE and Cross-Reference Table” on page 36.

If the suboption SHORT is specified, unreferenced identifiers are omitted.

The default suboption FULL means that FULL applies if you specify the option with no suboption.

If you specify both the XREF and ATTRIBUTES options, the two listings are combined. If there is a conflict between SHORT and FULL, the usage is determined by the last option specified. For example, ATTRIBUTES(SHORT) XREF(FULL) results in the FULL option for the combined listing.

---

## Input Record Formats

The compiler accepts both F-format and V-format records; the primary and secondary input data sets can have different formats.

The compiler determines the positions, within each record, of the PL/I source code and the sequence numbers from the following options:

Option	Specifying	IBM-supplied default
FMARGINS	Positions of source and sequence	FMARGINS(2,72)
FSEQUENCE	Numbers for F-format records	FSEQUENCE(73,80)
VMARGINS	Positions of source text and sequence	VMARGINS(10,100)
VSEQUENCE	Numbers for V-format records	VSEQUENCE(1,8)
MARGINS	Overriding values for above options	—
SEQUENCE	Overriding values for above options	—

You can set the values of FMARGINS, FSEQUENCE, VMARGINS and VSEQUENCE only when you install the compiler. If you do not set values at this time, the IBM-supplied default values apply. You can specify MARGINS and SEQUENCE when you invoke the compiler. When specified, they override either FMARGINS and FSEQUENCE or VMARGINS and VSEQUENCE, depending on whether the first input data set read by the syntax-checking stage of the compiler is F-format. The overriding values also apply if the compiler reads records of the same format as secondary input. If the records of the other format are read as the compiler installation values, the values for that format apply.

---

## Specifying Options in the %PROCESS or \*PROCESS statements

The compiler uses the %PROCESS statement to identify the start of each external procedure and to allow compile-time options to be specified for each compilation. The options you specify in adjacent %PROCESS statements apply to the compilation of the source statements to the end of input, or the next %PROCESS statement.

To specify options in the %PROCESS statement, code as follows:

```
%PROCESS options;
```

where *options* is a list of compile-time options. You must end the list of options with a semicolon, and the options list should not extend beyond the default right-hand source margin. The asterisk must appear in the first data byte of the record. If the records are F format, the asterisk must be in column 1. If the records are V or U format, the asterisk must be as far left as possible, that is column 1 if possible, or immediately following the sequence numbers if these are on the extreme left. The keyword %PROCESS can follow in the next byte (column) or after any number of blanks. You must separate option keywords by a comma or at least one blank.

The number of characters is limited only by the length of the record. If you do not wish to specify any options, code:

```
%PROCESS;
```

If you find it necessary to continue the %PROCESS statement onto the next record, terminate the first part of the list after any delimiter, and continue on the next record. You can split option keywords or keyword arguments when continuing onto

the next record, provided that the keyword or argument string terminates in the right-hand source margin, and the remainder of the string starts in the same column as the asterisk. You can continue a %PROCESS statement on several lines, or start a new %PROCESS statement. An example of multiple adjacent %PROCESS statements is as follows:

```
%PROCESS INT F(I) AG A(F) ESD MAP OP STG NEST X(F) SOURCE ;
%PROCESS LIST TEST ;
```

For information about using the %PROCESS statement with batched compilation, see “Compiling Multiple Procedures in a Single Job Step” on page 69.

Compile-time options, their abbreviated syntax, and their IBM-supplied defaults are shown in Table 3 on page 5 and Table 4 on page 7. Your site might have changed the IBM-supplied defaults or deleted options. Be sure to check for any changes before using compile-time option defaults. You can reinstate deleted compile-time options for a compilation by using the CONTROL compile-time option.

---

## Using the Preprocessor

The preprocessing facilities of the compiler are described in the *PL/I for MVS & VM Language Reference*. You can include statements in your PL/I program that, when executed by the preprocessor stage of the compiler, modify the source program or cause additional source statements to be included from a library. The following discussion provides some illustrations of the use of the preprocessor and explains how to establish and use source statement libraries.

## Invoking the Preprocessor

If you specify the compile-time option MACRO, the preprocessor stage of the compiler is executed. The compiler and the preprocessor use the data set defined by the DD statement with the name SYSUT1 during processing. They also use this data set to store the preprocessed source program until compilation begins. The IBM-supplied cataloged procedures for compilation include a DD statement with the name SYSUT1.

The format of the preprocessor output is given in Table 6.

---

*Table 6. Format of the Preprocessor Output*

Column 1	Printer control character, if any, transferred from the position specified in the MARGINS option.
Columns 2-72	Source program. If the original source program used more than 71 columns, additional lines are included for any lines that need continuation. If the original source program used fewer than 71 columns, extra blanks are added on the right.
Columns 73-80	Sequence number, right-aligned. If either SEQUENCE or NUMBER applies, this is taken from the sequence number field. Otherwise, it is a preprocessor generated number, in the range 1 through 99999. This sequence number will be used in the listing produced by the INSOURCE and SOURCE options, and in any preprocessor diagnostic messages.
Column 81	blank
Columns 82, 83	Two-digit number giving the maximum depth of replacement by the preprocessor for this line. If no replacement occurs, the columns are blank.
Column 84	E signifying that an error occurred while replacement was being attempted. If no error occurred, the column is blank.

---

Three other compile-time options, MDECK, INSOURCE, and SYNTAX, are meaningful only when you also specify the MACRO option. For more information about these options, see MDECK on page 17, INSOURCE on page 13, and SYNTAX on page 24.

A simple example of the use of the preprocessor to produce a source deck is shown in Figure 1. According to the value assigned to the preprocessor variable USE, the source statements will represent either a subroutine (CITYSUB) or a function (CITYFUN). The DSNAME used for SYSPUNCH specifies a source program library on which the preprocessor output will be placed. Normally compilation would continue and the preprocessor output would be compiled.

---

```

//OPT4#8 JOB
//STEP2 EXEC IELIC,PARM.PLI='MACRO,MDECK,NOCOMPILE,NOSYNTAX'
//PLI.SYSPUNCH DD DSNAME=HPU8.NEWLIB(FUN),DISP=(NEW,CATLG),UNIT=SYSDA,
//          SPACE=(TRK,(1,1,1)),DCB=(RECFM=FB,LRECL=80,BLKSIZE=400)
//PLI.SYSIN DD *
/* GIVEN ZIP CODE, FINDS CITY */
%DCL USE CHAR;
%USE = 'FUN' /* FOR SUBROUTINE, %USE = 'SUB' */ ;
%IF USE = 'FUN' %THEN %DO;
CITYFUN: PROC(ZIPIN) RETURNS(CHAR(16)) REORDER; /* FUNCTION */
          %END;
          %ELSE %DO;
CITYSUB: PROC(ZIPIN, CITYOUT) REORDER; /* SUBROUTINE */
          DCL CITYOUT CHAR(16); /* CITY NAME */
          %END;
DCL (LBOUND, HBOUND) BUILTIN;
DCL ZIPIN PIC '99999'; /* ZIP CODE */
DCL 1 ZIP_CITY(7) STATIC, /* ZIP CODE - CITY NAME TABLE */
      2 ZIP PIC '99999' INIT(
          95141, 95014, 95030,
          95051, 95070, 95008,
          0), /* WILL NOT LOOK AT LAST ONE */
      2 CITY CHAR(16) INIT(
          'SAN JOSE', 'CUPERTINO', 'LOS GATOS',
          'SANTA CLARA', 'SARATOGA', 'CAMPBELL',
          'UNKNOWN CITY'); /* WILL NOT LOOK AT LAST ONE */
DCL I FIXED BIN(31);
DO I = LBOUND(ZIP,1) TO /* SEARCH FOR ZIP IN TABLE */
      HBOUND(ZIP,1)-1 /* DON'T LOOK AT LAST ELEMENT */
      WHILE(ZIPIN ^= ZIP(I));
END;
%IF USE = 'FUN' %THEN %DO;
RETURN(CITY(I)); /* RETURN CITY NAME */
          %END;
          %ELSE %DO;
CITYOUT=CITY(I); /* RETURN CITY NAME */
          %END;
END;

```

---

Figure 1. Using the preprocessor to Produce a Source Deck That Is Placed on a Source Program Library

## Using the %INCLUDE Statement

The *PL/I for MVS & VM Language Reference* describes how to use the %INCLUDE statement to incorporate source text from a library into a PL/I program. (A *library* is an MVS partitioned data set or a VM MACLIB that can be used to store other data sets called members.) Source text that you might want to insert into a PL/I program using a %INCLUDE statement must exist as a member within a library. “Source Statement Library (SYSLIB)” on page 67 further describes the process of defining a source statement library to the compiler.

The statement:

```
%INCLUDE DD1 (INVERT);
```

specifies that the source statements in member INVERT of the library defined by the DD statement with the name DD1 are to be inserted consecutively into the source program. The compilation job step must include appropriate DD statements.

If you omit the ddname, the ddname SYSLIB is assumed. In such a case, you must include a DD statement with the name SYSLIB. (The IBM-supplied cataloged procedures do not include a DD statement with this name in the compilation procedure step.)

A %PROCESS statement in source text included by a %INCLUDE statement results in an error in the compilation.

Figure 2 shows the use of a %INCLUDE statement to include the source statements for FUN in the procedure TEST. The library HPU8.NEWLIB is defined in the DD statement with the qualified name PLI.SYSLIB, which is added to the statements of the cataloged procedure IEL1CLG for this job. Since the source statement library is defined by a DD statement with the name SYSLIB, the %INCLUDE statement need not include a ddname.

It is not necessary to invoke the preprocessor if your source program, and any text to be included, does not contain any macro statements. Under these circumstances, you can obtain faster inclusion of text by specifying the INCLUDE compile-time option.

---

```
//OPT4#9      JOB
//STEP3      EXEC IEL1CLG,PARM.PLI='INC,S,A,X,NEST'
//PLI.SYSLIB DD DSN=HPU8.NEWLIB,DISP=OLD
//PLI.SYSIN DD *
TEST: PROC OPTIONS(MAIN) REORDER;
  DCL ZIP PIC '99999';          /* ZIP CODE          */
  DCL EOF BIT INIT('0'B);
  ON ENDFILE(SYSIN) EOF = '1'B;
  GET EDIT(ZIP) (COL(1), P'99999');
  DO WHILE(-EOF);
    PUT SKIP EDIT(ZIP, CITYFUN(ZIP)) (P'99999', A(16));
    GET EDIT(ZIP) (COL(1), P'99999');
  END;
  %PAGE;
  %INCLUDE FUN;
END;                          /* TEST          */
//GO.SYSIN DD *
95141
95030
94101
//
```

---

Figure 2. Including Source Statements from a Library

## Using the PL/I Preprocessor in Program Testing

You can use the %INCLUDE PL/I preprocessor statement to include program-testing statements from the source statement library in your program when you test it. You can use these statements in conjunction with program checkout statements to help track your program's operation and handle errors that occur.

---

## Using % Statements

Statements that direct the operation of the compiler, begin with a percent (%) symbol. These statements must not have label or condition prefixes, and cannot be a "unit" of a compound statement.

The % statements allow you to control the source program listing and to include external strings in the source program. These control statements, %INCLUDE, %PRINT, %NOPRINT, %PAGE, and %SKIP, are listed below and described fully in the *PL/I for MVS & VM Language Reference*.

%INCLUDE	Directs the compiler to incorporate external strings of characters and/or graphics into the source program.
%PRINT	Directs the compiler to resume printing the source and insource listings.
%NOPRINT	Directs the compiler to suspend printing the source and insource listings until a %PRINT statement is encountered.
%PAGE	Directs the compiler to print the statement immediately after a %PAGE statement in the program listing on the first line of the next page.
%SKIP	Specifies the number of lines to be skipped.

**Note:** You should place each % statement on a line by itself.

---

## Invoking the Compiler from an Assembler Routine

You can invoke the compiler from an assembler language program by using one of the macro instructions ATTACH, CALL, LINK, or XCTL. The following information supplements the description of these macro instructions given in the supervisor and data management manual.

You cannot dynamically invoke the compiler under VM from an assembler routine running in a user area.

To invoke the compiler specify IEL1AA as the entry point name.

You can pass three address parameters to the compiler:

1. The address of a compile-time option list
2. The address of a list of ddnames for the data sets used by the compiler
3. The address of a page number that is to be used for the first page of the compiler listing on SYSPRINT

These addresses must be in adjacent fullwords, aligned on a fullword boundary. Register 1 must point to the first address in the list, and the first (left-hand) bit of the last address must be set to 1, to indicate the end of the list.

**Note:** If you want to pass parameters in an XCTL macro instruction, you must use the execute (E) form of the macro instruction. Remember also that the XCTL macro instruction indicates to the control program that the load module containing the XCTL macro instruction is completed. Thus the parameters must be established in a portion of main storage outside the load module containing the XCTL macro instruction, in case the load module is deleted before the compiler can use the parameters.

The format of the three parameters for all the macro instructions is described below.

## Option List

The option list must begin on a halfword boundary. The first two bytes contain a binary count of the number of bytes in the list (excluding the count field). The remainder of the list can comprise any of the compile-time option keywords, separated by one or more blanks, a comma, or both of these.

## DDNAME List

The ddname list must begin on a halfword boundary. The first two bytes contain a binary count of the number of bytes in the list (excluding the count field). Each entry in the list must occupy an 8-byte field; the sequence of entries is given in Table 7.

If a ddname is shorter than 8 bytes, fill the field with blanks on the right. If you omit an entry, fill its field with binary zeros; however, you can omit entries at the end of the list entirely.

---

*Table 7. Entry Dequence in the DDNAME List*

---

<b>Entry</b>	<b>Standard DDNAME</b>
1	SYSLIN
2	not applicable
3	not applicable
4	SYSLIB
5	SYSIN
6	SYSPRINT
7	SYSPUNCH
8	SYSUT1
9	not applicable
10	not applicable
11	not applicable
12	not applicable
13	not applicable
14	SYSCIN

---

## Page Number

The compiler adds 1 to the last page number used in the compiler listing and put this value in the page-number field before returning control to the invoking routine. Thus, if the compiler is invoked again, page numbering is continuous.

---

## Using the Compiler Listing

During compilation, the compiler generates a listing, most of which is optional, that contains information about the source program, the compilation, and the object module. It places this listing in the data set defined by the DD statement with the name SYSPRINT (usually output to a printer). In a conversational environment, you can also request a listing at your terminal (using the TERMINAL option). The following description of the listing refers to its appearance on a printed page.

The first part of Table 4 on page 7 shows the components that can be included in the compiler listing. The rest of this section describes them in detail.

Of course, if compilation terminates before reaching a particular stage of processing, the corresponding listings do not appear.

The listing comprises a small amount of standard information that always appears, together with those items of optional information specified or supplied by default. The listing at the terminal contains only the optional information that has been requested in the TERMINAL option.

## Heading Information

The first page of the listing is identified by the product number, the compiler version number, and the date and the time compilation commenced. This page and subsequent pages are numbered.

Near the end of the listing you will find either a statement that no errors or warning conditions were detected during the compilation, or a message that one or more errors were detected. The format of the messages is described under "Messages" on page 44. The second to the last line of the listing shows the CPU time taken for the compilation. The last line of the listing is "END OF COMPILATION OF xxxx" where "xxxx" is the external procedure name. If you specify the NOSYNTAX compile-time option, or the compiler aborts early in the compilation, the external procedure name "xxxx" is not included and the line truncates to "END OF COMPILATION."

The following paragraphs describe the optional parts of the listing in the order in which they appear.

## Options Used for the Compilation

If the option OPTIONS applies, a complete list of the options specified for the compilation, including the default options, appears on the first page.

## Preprocessor Input

If both the options MACRO and INSOURCE apply, the compiler lists input to the preprocessor, one record per line, each line numbered sequentially at the left.

If the preprocessor detects an error, or the possibility of an error, it prints a message on the page or pages following the input listing. The format of these messages is the same as the format for the compiler messages described under "Messages" on page 44.

## SOURCE Program

If the option SOURCE applies, the input to the compiler is listed, one record per line. If the input records contain printer control characters or %SKIP or %PAGE statements, the lines are spaced accordingly. You can use %NOPRINT and %PRINT statements to stop and restart the printing of the listing.

If the MACRO option applies, the source listing shows the included text in place of the %INCLUDE statements in the primary input data set.

If the MACRO option does not apply but the INCLUDE option does, the included text is bracketed by comments indicating the %INCLUDE statement that caused the text to be included. Each nested %INCLUDE has the comment text indented two positions to the right.

Assume the following source input on SYSIN:

```
MAIN: PROC REORDER;  
%INCLUDE MEMBER1;  
END;
```

and the following content of MEMBER1:

```
J=K;  
%INCLUDE DSALIB1(DECLARES);  
L=M;
```

and the following content of DECLARES:

```
DCL (NULL,DATE) BUILTIN;
```

produces in the source listing:

```
MAIN: PROC REORDER;  
/*BEGIN %INCLUDE SYSLIB (MEMBER1 )*****/  
J=K;  
/**BEGIN %INCLUDE DSALIB1 (DECLARES)*****/  
DCL (NULL,DATE) BUILTIN;  
/**END %INCLUDE DSALIB1 (DECLARES)*****/  
L=M;  
/*END %INCLUDE SYSLIB (MEMBER1 )*****/  
END;
```

If the STMT compile-time option applies, the statement numbers are derived from a count of the number of statements in the program after %INCLUDEs have been processed.

If the NUMBER option applies, the compiler derives statement numbers from the sequence numbers of the statements in the source records after %INCLUDE statements have been processed. Normally the compiler uses the last five digits as statement numbers. If, however, this does not produce a progression of statements with successively higher numbers, the compiler adds 100000 to all statement numbers starting from the one that would otherwise be equal to or less than its predecessor.

For instance, if a V-format primary input data set had the following lines:

```
00001000 A:PROC;  
00002000 %INCLUDE B;  
00003000 END;
```

and member B contained:

```
00001000 C=D;  
00002000 E=F;  
00003000 G=H;
```

then the source listing would be as follows:

```
SOURCE LISTING
NUMBER

    1000  00001000 A:PROC;
           00002000      /*BEGIN %INCLUDE SYSLIB (B      )*****/
101000  00001000 C=D;
102000  00002000 E=F;
103000  00003000 G=H;
           /*END %INCLUDE SYSLIB (B      )*****/
203000  00003000 END;
```

The additional 100000 has been introduced into the statement numbers at two points:

1. Beginning at the first statement of the included text (the statement C=D;)
2. Beginning with the first statement after the included text (the END statement)

If the source statements are generated by the preprocessor, columns 82-84 contain diagnostic information, as shown in Table 6 on page 29.

## Statement Nesting Level

If the option NEST applies, the block level and the DO-level are printed to the right of the statement or line number under the headings LEV and NT respectively, for example:

```
STMT  LEV  NT
      1      0  A: PROC OPTIONS(MAIN);
      2      1  0  B: PROC;
      3      2  0      DCL K(10,10) FIXED BIN (15);
      4      2  0      DCL Y FIXED BIN (15) INIT (6);
      5      2  0      DO I=1 TO 10;
      6      2  1          DO J=1 TO 10;
      7      2  2              K(I,J) = N;
      8      2  2          END;
      9      2  1      BEGIN;
     10      3  1          K(1,1)=Y;
     11      3  1      END;
     12      2  1  END B;
     13      1  0  END A;
```

## ATTRIBUTE and Cross-Reference Table

If the option ATTRIBUTES applies, the compiler prints an attribute table containing a list of the identifiers in the source program together with their declared and default attributes. In this context, the attributes include any relevant options, such as REFER, and also descriptive comments, such as:

```
/*STRUCTURE*/
```

If the option XREF applies, the compiler prints a cross-reference table containing a list of the identifiers in the source program together with the numbers of the statements in which they appear. If both ATTRIBUTES and XREF apply, the two tables are combined. If the suboption SHORT applies, unreferenced identifiers are not listed.

If the following conditions apply:

- GRAPHIC compile-time option is in effect
- Compilation is being done under MVS
- At least one DBCS identifier is found in the compilation unit
- ATTRIBUTES and/or XREF are in effect

then the PL/I compiler uses the DBCS Ordering Support Product to perform the sorting of the DBCS identifiers for the XREF listing.

The types of ordering available are the Total Stroke Count (KS), Radical Stroke Count (KR), and the IBM Unique Pronunciation (KU). The default is KU. To select the other types you must supply a special AKSLDFLT CSECT specifying the desired ordering type.

All sorted DBCS identifiers appear in the listing before the SBCS identifiers, which are sorted in collating sequence.

The DBCSOS Ordering Product requires 128K of free storage. For information about reserving storage, see the SIZE option, "Under MVS" on page 23.

### **Attribute Table**

If you declare an identifier explicitly, the compiler lists the number of the DECLARE statement. The compiler indicates an undeclared variable by asterisks. (The compiler also lists undeclared variables in error messages.) It also gives the statement numbers of statement labels and entry labels.

The compiler never includes the attributes INTERNAL and REAL. You can assume them unless the respective conflicting attributes, EXTERNAL and COMPLEX, appear.

For a file identifier, the attribute FILE always appears, and the attribute EXTERNAL appears if it applies; otherwise, the compiler only lists explicitly declared attributes.

The compiler prints the dimension attribute for an array first. It prints the bounds as in the array declaration, but it replaces expressions with asterisks. Structure levels other than base elements also have their bounds replaced by asterisks.

For a character string or a bit string, the compiler prints the length, preceded by the word BIT or CHARACTER, as in the declaration, but it replaces an expression with an asterisk.

### **Cross-Reference Table**

If you combine the cross-reference table with the attribute table, the numbers of the statements or lines in which a name appears follow the list of attributes for the name. The order in which the statement numbers appear is subject to any reordering of blocks that has occurred during compilation. In general, the compiler gives the statement numbers for the outermost block first, followed on the next line by the statement numbers for the inner blocks.

The compiler expands and optimizes PL/I text before it produces the cross-reference table. Consequently, some names that appear only once within a source statement can acquire multiple references to the same statement number. By the same token, other names can appear to have incomplete lists of references,

while still others can have references to statements in which the name does not appear explicitly.

For example:

- Duplicate references can be listed for items such as do-loop control variables, and for some aggregates.
- Optimization of certain operations on structures can result in incomplete listings in the cross-reference table. The numbers of statements in which these operations are performed on major or minor structures are listed against the names of the elements, instead of against the structure names.
- No references to PROCEDURE or ENTRY statements in which a name appears as a parameter are listed in the cross-reference table entry for that name.
- References within DECLARE statements to variables that are not being declared are not listed. For example, in the statements:

```
DCL ARRAY(N);  
DCL STRING CHAR(N);
```

no references to these statements would appear in the cross-reference table entry for N.

- The number of a statement in which an implicitly pointer-qualified based variable name appears is included not only in the list of statement numbers for that name, but also in the list of statement numbers for the pointer implicitly associated with it.
- The statement number of an END or LEAVE statement that refers to a label is not listed in the entry for the label.
- Automatic variables declared with the INITIAL attribute have a reference to the PROCEDURE or BEGIN statement for the block containing the declaration included in the list of statement numbers.

## Aggregate Length Table

An aggregate length table is obtained by using the AGGREGATE option. The table shows how the compiler maps each aggregate in the program. It contains the following information:

- The statement number in which the aggregate is declared.
- The name of the aggregate and the element within the aggregate.
- The level number of each item in a structure.
- The number of dimensions in an array.
- The byte offset of each element from the beginning of the aggregate. (The compiler does not give bit offsets for unaligned bit-string data). As a word of caution, be careful when interpreting the data offsets indicated in the data length table. An odd offset does not necessarily represent a data element without halfword, fullword, or even double word alignment. If you specify or infer the aligned attribute for a structure or its elements, the proper alignment requirements are consistent with respect to other elements in the structure, even though the table does not indicate the proper alignment relative to the beginning of the table.
- The length of each element.

- The total length of each aggregate, structure, and substructure.

If there is padding between two structure elements, a */\*PADDING\*/* comment appears, with appropriate diagnostic information.

The table is completed with the sum of the lengths of all aggregates that do not contain adjustable elements.

The statement or line number identifies either the DECLARE statement for the aggregate, or, for a controlled aggregate, an ALLOCATE statement for the aggregate. An entry appears for each ALLOCATE statement involving a controlled aggregate, as such statements can have the effect of changing the length of the aggregate during run time. Allocation of a based aggregate does not have this effect, and only one entry, which is that corresponding to the DECLARE statement, appears.

When passing an aggregate to a subroutine, the length of an aggregate might not be known during compilation, either because the aggregate contains elements having adjustable lengths or dimensions, or because the aggregate is dynamically defined. In these cases, the compiler prints the word *adjustable* or *defined* in the *offset* column and *param* for parameter in the *element length* and *total length* columns. Because the compiler might not know the length of an aggregate during compilation, it does not print padding information.

An entry for a COBOL mapped structure has the word COBOL appended. COBOL mapped structures are structures into which a program reads or writes a COBOL record, or a structure that can be passed between PL/I programs and COBOL programs. The COBOL entry appears if the compiler determines that the COBOL and PL/I mapping for the structure is different, and the creation of a temporary structure mapped according to COBOL synchronized structure rules is not suppressed by NOMAP, NOMAPIN, or NOMAPOUT.

If a COBOL entry does appear it is additional to the entry for the PL/I mapped version of the structure.

The compiler makes a separate entry in the aggregate table for every aggregate dummy argument or COBOL mapped structure.

## Storage Requirements

If the option STORAGE applies, the compiler lists the following information under the heading *Storage Requirements* on the page following the end of the aggregate length table:

- The length of the program control section. The program control section is the part of the object that contains the executable part of the program.
- The length of the static internal control section. This control section contains all storage for variables declared STATIC INTERNAL.
- The storage area in bytes for each procedure.
- The storage area in bytes for each begin-block.
- The storage area in bytes for each ON-unit.
- The dynamic storage area in bytes for each procedure, begin-block, and ON-unit. The dynamic storage area is acquired at activation of the block.

## Statement Offset Addresses

If the option LIST applies, the compiler includes a pseudo-assembler listing in the compiler listing. You can use the offset given in run-time error messages to discover the erroneous statement, because the offsets in both run-time messages and the pseudo-assembler listing are relative to the start of the external procedure. Simply match the offset given in the error message with the offset in the listing to find the erroneous statement.

In the example shown in Figure 3, compile unit offset +17E occurs in the object listing under statement 6. Statement 6 is the erroneous statement.

---

```
                SOURCE LISTING

1  M:PROC OPTIONS(MAIN);
2  CALL A2;
3  A1:PROC;
4  N=3;
5  A2:ENTRY;
6  N=N/0;
7  END;
8  END;

- OBJECT LISTING

* STATEMENT NUMBER 6
00016C 58 70 D 0C0          L    7,192(0,13)
000170 48 60 3 02A          LH   6,42(0,3)
000174 48 80 7 0B8          LH   8,N
000178 1B 99                SR   9,9
00017A 8E 80 0 010          SRDA 8,16
00017E 1D 86                DR   8,6
000180 12 99                LTR  9,9
000182 47 B0 2 02A          BNM  CL.13
000186 5A 90 3 034          A    9,52(0,3)
00018A                    CL.13 EQU  *
00018A 8A 90 0 010          SRA  9,16
00018E 40 90 7 0B8          STH  9,N

Message:

IBM0301S ONCODE=320 The ZERODIVIDE condition was raised.
                From compile unit M at entry point A2 at compile
                unit offset +0000017E at address 000201FE.
```

---

*Figure 3. Finding Statement Number from a Compile Unit Offset in an Error Message*

If the OFFSET option applies, the compiler lists for each primary entry point the offsets at which statements occur. This information is found in the compiler listing under the heading, "Table of Offsets and Statement Numbers."

Entry offsets given in dump and on-unit SNAP error messages can be compared with this table and the erroneous statement discovered. The statement is identified by finding the section of the table that relates to the block named in the message and then finding the largest offset less than or equal to the offset in the message. The statement number associated with this offset is the one needed.

If a secondary entry point is used, first find the name of the block that contains this entry and the corresponding section of the offset table that relates to this name. Next, add the offset given in the message to the offset of the secondary entry point in the table. This will convert the message offset so that it is relative to the primary entry point versus the secondary entry point, which was entered during execution.

Use this converted offset to search the section of the offset table for the largest offset as described above.

In the example in Figure 4, secondary entry point P2 is contained in procedure block P1 at offset X'78'. Adding X'78' to the message entry offset of X'44' yields a value of X'BC'. The largest offset table entry less than or equal to X'BC' is X'B4', which corresponds to statement number 7.

---

STMT	SOURCE LISTING				
1	Q:	PROC	OPTIONS(MAIN);		
2		ON	ERROR	SNAP	GOTO L;
3		CALL	P2;		
4	P1:	PROC;			
5		N=1;			
6	P2:	ENTRY;			
7		SIGNAL	ERROR;		
8		END;			
9	L:	END;			

TABLE OF OFFSETS AND STATEMENT NUMBERS					
WITHIN PROCEDURE Q					
OFFSET (HEX)	0	A8	C0	CA	
STATEMENT NO.	1	2	3	9	
WITHIN PROCEDURE P1					
OFFSET (HEX)	0	78	A8	B4	BE
STATEMENT NO.	4	6	5	7	8

Messages:

```
'ERROR' condition was raised
Traceback of user routines:
Compile Unit  Entry Statement  CU offset  Entry offset  Address
Q             P2             +000001A0 +00000044    00020220
Q             Q             +000000CC +000000C8    0002014C
```

---

Figure 4. Finding Statement Number from an Entry Offset in an Error Message

## External Symbol Dictionary

If the option ESD applies, the compiler lists the contents of the external symbol dictionary (ESD).

The ESD is a table containing all the external symbols that appear in the object module. (The machine instructions in the object module are grouped together in *control sections*; an external symbol is a name that can be referred to in a control section other than the one in which it is defined.) The contents of an ESD appear under the following headings:

- SYMBOL    An 8-character field that identifies the external symbol.
- TYPE      Two characters from the following list to identify the type of entry:
  - SD        Section definition: the name of a control section within the object module.
  - CM        Common area: a type of control section that contains no data or executable instructions.

ER	External reference: an external symbol that is not defined in the object module.
WX	Weak external reference: an external symbol that is not defined in this module and that is not to be resolved unless an ER entry is encountered for the same reference.
PR	Pseudoregister: a field used to address files, controlled variables, and FETCHed procedures.
LD	Label definition: the name of an entry point to the external procedure other than that used as the name of the program control section.
ID	Four-digit hexadecimal number: all entries in the ESD, except LD-type entries, are numbered sequentially, beginning with 0001.
ADDRESS	Hexadecimal representation of the address of the external symbol.
LENGTH	The hexadecimal length in bytes of the control section (SD, CM and PR entries only).

### ESD Entries

The external symbol dictionary usually starts with the standard entries shown in Figure 5, which assumes the existence of an external procedure called NAME.

SYMBOL	TYPE	ID	ADDRESS	LENGTH
CEESTART	SD	0001	000000	000080
***NAME1	SD	0002	000000	0000A8
***NAME2	SD	0003	000000	00005C
CEEMAIN	WX	0004	000000	
CEEMAIN	SD	0005	000000	000010
IBMRINP1	ER	0006	000000	
CEEFMAIN	WX	0007	000000	
CEEBETBL	ER	0008	000000	
CEERootA	ER	0009	000000	
CEESG010	ER	000A	000000	
NAME	LD		000008	

Figure 5. External Symbol Dictionary

#### \*\*\*name1

SD-type entry for the program control section (the control section that contains the executable instructions of the object module). This name is the first label of the external procedure, padded on the left with asterisks to 7 characters if necessary, and extended on the right with the character 1.

#### \*\*\*name2

SD-type entry for the static internal control section (which contains main storage for all variables declared STATIC INTERNAL). This name is the first label of the external procedure, padded on the left with asterisks to 7 characters if necessary, and extended on the right with the character 2.

#### CEESTART

SD-type entry for CEESTART. This control section transfers control to CEERootA, the initialization routine for the library environment. When initialization is complete, control passes to the address stored in the control section CEEMAIN. (Initialization is required only once while a PL/I program is running, even if it calls another external procedure. In such a case, control

passes directly to the entry point named in the CALL statement, and not to the address contained in CEEMAIN.)

### **CEEROOTA, CEESG010, CEEBETBL, IBMRINP1**

These ER-type entries are generated to support environment initialization for the program.

The other entries in the external symbol dictionary vary, but can include the following:

- SD-type entry for the control section CEEMAIN, which contains the address of the primary entry point to the external procedure. This control section is present only if the procedure statement includes the option MAIN. A WX-type entry for CEEMAIN is always generated to support environment initialization for the program.
- Reference to a number of control sections as follows:

CEEFMAIN	A control section used in <i>fetch</i> processing. It indicates the presence of a fetchable entry point within the load module.
IBMSEATA	A module in the PL/I library used to set the attention exit for use in procedures compiled with the INTERRUPT option. This is an ER type entry if the procedure was compiled with the INTERRUPT option.
CEEUOPT	A control section that contains the run-time options specified at compile time.
PLIXOPT	Run-time options string control section.
- LD-type entries for all names of entry points to the external procedure.
- ER-type entries for all the library subroutines and external procedures called by the source program.
- CM-type entries for variables declared STATIC EXTERNAL without the INITIAL attribute.
- SD-type entries for all other STATIC EXTERNAL variables and for external file names.
- PR-type entries for all file names. For external file names, the name of the pseudoregister is the same as the file name; for internal file names, the compiler generates pseudoregister names.
- PR-type entries for all controlled variables. For external variables, the name of the variable is used for the pseudoregister name; for internal variables, the compiler generates names.
- PR-type entries for fetched entry names.

## **Static Internal Storage Map**

The MAP option produces a Variable Offset Map. This map shows how PL/I data items are mapped in main storage. It names each PL/I identifier, its level, its offset from the start of the storage area in both decimal and hexadecimal form, its storage class, and the name of the PL/I block in which it is declared.

If the LIST option is also specified a map of the static internal and external control sections is also produced.

For more information about the static internal storage map and an example, see the *Language Environment for MVS & VM Debugging Guide and Run-Time Messages*.

## Object Listing

If the option LIST applies, the compiler generates a listing of the machine instructions of the object module, including any compiler-generated subroutines, in a form similar to assembler language.

For more information about the object listing and an example, see the *Language Environment for MVS & VM Debugging Guide and Run-Time Messages*.

## Messages

If the preprocessor or the compiler detects an error, or the possibility of an error, they generate messages. Messages generated by the preprocessor appear in the listing immediately after the listing of the statements processed by the preprocessor. You can generate your own messages in the preprocessing stage by use of the %NOTE statement. Such messages might be used to show how many times a particular replacement had been made. Messages generated by the compiler appear at the end of the listing. All messages are graded according to their severity, as follows:

- I An information message that calls attention to a possible inefficiency in the program or gives other information generated by the compiler.
- W A warning message that calls attention to a possible error, although the statement to which it refers is syntactically valid.
- E An error message that describes an error detected by the compiler for which the compiler applied a *fix-up* with confidence. The resulting program will run, and it will probably give correct results.
- S A severe error message that specifies an error detected by the compiler for which the compiler cannot apply a *fix-up* with confidence. The resulting program will run but will not give correct results.
- U An unrecoverable error message that describes an error that forces termination of the compilation.

The compiler only lists messages that have a severity equal to or greater than that specified by the FLAG option, as shown in Table 8 on page 45.

Each message is identified by an eight-character code of the form *IELnnnnI*, where:

- The first three characters *IEL* identify the message as coming from the compiler.
- The next four characters, *nnnn*, are a four-digit message number.
- The last character, *I*, is an operating system code for the operator indicating that the message is for information only.

The text of each message, an explanation, and any recommended programmer response, are given in the *PL/I for MVS & VM Compile-Time Messages and Codes*.

*Table 8. Using the FLAG Option To Select the Lowest Message Severity Listed*

Type of Message	Option
Information	FLAG(I)
Warning	FLAG(W)
Error	FLAG(E)
Severe Error	FLAG(S)
Unrecoverable Error	Always listed

## Return Codes

For every compilation job or job step, the compiler generates a return code that indicates to the operating system the degree of success or failure it achieved. For MVS, this code appears in the *end-of-step* message that follows the listing of the job control statements and job scheduler messages for each step. The meaning of the codes are given in Table 9.

*Table 9. Return Codes from Compilation of a PL/I Program*

Return code	Description
0000	No error detected; compilation completed, successful execution anticipated.
0004	Warning; possible error detected; compilation completed, execution probable.
0008	Error detected; compilation completed; successful execution probable.
0012	Severe error detected; compilation not necessarily completed; successful execution improbable.
0016	Unrecoverable error detected; compilation terminated abnormally; successful execution impossible.

---

## Chapter 2. Using PL/I Cataloged Procedures under MVS

This chapter describes the standard cataloged procedures supplied by IBM for use with the IBM PL/I for MVS & VM compiler. It explains how to invoke them, and how to temporarily or permanently modify them. You must be linked to Language Environment before using any of the cataloged procedures described in this chapter.

A cataloged procedure is a set of job control statements stored in a library. A cataloged procedure includes one or more EXEC statements, each of which can be followed by one or more DD statements. You can retrieve the statements by naming the cataloged procedure in the PROC parameter of an EXEC statement in the input stream.

You can use cataloged procedures to save time and reduce Job Control Language errors. If the statements in a cataloged procedure do not match your requirements exactly, you can easily modify them or add new statements for the duration of a job. You should review these procedures and modify them to obtain the most efficient use of the facilities available and to allow for your own conventions.

---

### IBM-Supplied Cataloged Procedures

The PL/I cataloged procedures supplied for use with the IBM PL/I for MVS & VM are:

<b>IEL1C</b>	Compile only
<b>IEL1CL</b>	Compile and link-edit
<b>IEL1CLG</b>	Compile, link-edit, and run
<b>IEL1CG</b>	Compile, load and run

The information in this section describes the procedure steps of the different cataloged procedures. For a description of the individual statements for compiling and link editing, see "Using JCL during Compilation" on page 64 and the *Language Environment for MVS & VM Programming Guide*. These cataloged procedures do not include a DD statement for the input data set; you must always provide one.

The example shown in Figure 6 on page 47 illustrates the JCL statements you might use to invoke the cataloged procedure IEL1CLG to compile, link-edit, and run a PL/I program.

**Note:** The IBM PL/I for MVS & VM requires a minimum REGION size of 512K. Large programs require more storage. If you do not specify REGION on the EXEC statement that invokes the cataloged procedure you are running, the compiler uses the default REGION size for your site. The default size might or might not be adequate, depending on the size of your PL/I program. For an example of specifying REGION on the EXEC statement, see Figure 6 on page 47.

---

```

//COLEGO    JOB
//STEP1     EXEC IEL1CLG, REGION.PLI=IM
//PLI.SYSIN DD *
            .
            .
            .
            (insert PL/I program to be compiled here)
            .
            .
            .
/*

```

---

Figure 6. Invoking a Cataloged Procedure

## Compile Only (IEL1C)

This cataloged procedure, shown in Figure 7 on page 48, includes only one procedure step, in which the options specified for the compilation are OBJECT and NODECK. (IEL1AA is the symbolic name of the compiler.) In common with the other cataloged procedures that include a compilation procedure step, IEL1C does not include a DD statement for the input data set; you must always supply an appropriate statement with the qualified ddname PLI.SYSIN.

The OBJECT option causes the compiler to place the object module, in a syntax suitable for input to the linkage editor, in the standard data set defined by the DD statement with the name SYSLIN. This statement defines a temporary data set named &&LOADSET on a sequential device; if you want to retain the object module after the end of your job, you must substitute a permanent name for &&LOADSET (that is, a name that does not start with &&) and specify KEEP in the appropriate DISP parameter for the last procedure step that used the data set. You can do this by providing your own SYSLIN DD statement, as shown below. The data set name and disposition parameters on this statement will override those on the IEL1C procedure SYSLIN DD statement. In this example, the compile step is the only step in the job.

```

//PLICOMP EXEC IEL1C
//PLI.SYSLIN DD DSN=MYPROG,DISP=(MOD,KEEP)
//PLI.SYSIN DD ...

```

The term MOD in the DISP parameter in Figure 7 on page 48 allows the compiler to place more than one object module in the data set, and PASS ensures that the data set is available to a later procedure step providing a corresponding DD statement is included there.

The SYSLIN SPACE parameter allows an initial allocation of 250 eighty-byte records and, if necessary, 15 further allocations of 100 records (a total of 1750 records).

---

```

//IEL1C  PROC LNGPRFX='IEL.V1R1M1',LIBPRFX='CEE.V1R4M0',          00010000
//          SYSLBLK=3200                                         00020000
//*                                               00030000
//***** 00040000
//*                                               * 00050000
//* LICENSED MATERIALS - PROPERTY OF IBM                * 00060000
//*                                               * 00070000
//* 5688-235 (C) COPYRIGHT IBM CORP. 1964, 1995        * 00080000
//* ALL RIGHTS RESERVED                                  * 00090000
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,       * 00100000
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA        * 00110000
//* ADP SCHEDULE CONTRACT WITH IBM CORP.               * 00120000
//*                                               * 00130000
//* SEE COPYRIGHT INSTRUCTIONS                          * 00140000
//*                                               * 00150000
//***** 00160000
//*                                               00170000
//* IBM PL/I FOR MVS & VM                                  00180000
//*                                               00190000
//* COMPILE A PL/I PROGRAM                                00200000
//*                                               00210000
//* RELEASE LEVEL: 01.01.01 (VERSION.RELEASE.MODIFICATION LEVEL) 00220000
//*                                               00230000
//* PARAMETER  DEFAULT VALUE  USAGE                    00240000
//* LNGPRFX   IEL.V1R1M1      PREFIX FOR LANGUAGE DATA SET NAMES 00250000
//* LIBPRFX   CEE.V1R4M0      PREFIX FOR LIBRARY DATA SET NAMES  00260000
//* SYSLBLK   3200            BLKSIZE FOR OBJECT DATA SET        00270000
//*                                               00280000
//PLI      EXEC PGM=IEL1AA,PARM='OBJECT,NODECK',REGION=512K      00290000
//STEPLIB DD DSN=&LNGPRFX..SIELCOMP,DISP=SHR                    00300000
//        DD DSN=&LIBPRFX..SCEERUN,DISP=SHR                      00310000
//SYSPRINT DD SYSOUT=*                                           00320000
//SYSLIN  DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,           00330000
//        SPACE=(80,(250,100)),DCB=(BLKSIZE=&SYSLBLK)           00340000
//SYSUT1  DD DSN=&&SYSUT1,UNIT=SYSDA,                             00350000
//        SPACE=(1024,(200,50)),,CONTIG,ROUND),DCB=BLKSIZE=1024 00360000

```

---

Figure 7. Cataloged Procedure IEL1C

## Compile and Link-Edit (IEL1CL)

This cataloged procedure, shown in Figure 8 on page 49, includes two procedure steps: PLI, which is identical to cataloged procedure IEL1C, and LKED, which invokes the linkage editor (symbolic name IEWL) to link-edit the object module produced in the first procedure step.

Input data for the compilation procedure step requires the qualified ddname PLI.SYSIN. The COND parameter in the EXEC statement LKED specifies that this procedure step should be bypassed if the return code produced by the compiler is greater than 8 (that is, if a severe or unrecoverable error occurs during compilation).

```

//IEL1CL PROC LNGPRFX='IEL.V1R1M1',LIBPRFX='CEE.V1R4M0',          00010000
//          SYSLBLK=3200,GOPGM=GO                                00020000
//*                                                    00030000
//*****                                                    00040000
//*                                                    * 00050000
//* LICENSED MATERIALS - PROPERTY OF IBM                    * 00060000
//*                                                    * 00070000
//* 5688-235 (C) COPYRIGHT IBM CORP. 1964, 1995            * 00080000
//* ALL RIGHTS RESERVED                                      * 00090000
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,           * 00100000
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA            * 00110000
//* ADP SCHEDULE CONTRACT WITH IBM CORP.                   * 00120000
//*                                                    * 00130000
//* SEE COPYRIGHT INSTRUCTIONS                              * 00140000
//*                                                    * 00150000
//*****                                                    00160000
//*                                                    00170000
//* IBM PL/I FOR MVS & VM                                     00180000
//*                                                    00190000
//* COMPILE AND LINK EDIT A PL/I PROGRAM                   00200000
//*                                                    00210000
//* RELEASE LEVEL: 01.01.01 (VERSION.RELEASE.MODIFICATION LEVEL) 00220000
//*                                                    00230000
//* PARAMETER DEFAULT VALUE USAGE                          00240000
//* LNGPRFX IEL.V1R1M1 PREFIX FOR LANGUAGE DATA SET NAMES 00250000
//* LIBPRFX CEE.V1R4M0 PREFIX FOR LIBRARY DATA SET NAMES 00260000
//* SYSLBLK 3200 BLKSIZE FOR OBJECT DATA SET              00270000
//* GOPGM GO MEMBER NAME FOR LOAD MODULE                   00280000
//*                                                    00290000
//PLI EXEC PGM=IEL1AA,PARM='OBJECT,NODECK',REGION=512K      00300000
//STEPLIB DD DSN=&LNGPRFX..SIELCOMP,DISP=SHR               00310000
// DD DSN=&LIBPRFX..SCEERUN,DISP=SHR                       00320000
//SYSPRINT DD SYSOUT=*                                     00330000
//SYSLIN DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,       00340000
// SPACE=(80,(250,100)),DCB=(BLKSIZE=&SYSLBLK)             00350000
//SYSUT1 DD DSN=&&SYSUT1,UNIT=SYSDA,                        00360000
// SPACE=(1024,(200,50)),,CONTIG,ROUND),DCB=BLKSIZE=1024 00370000
//LKED EXEC PGM=IEWL,PARM='XREF,LIST',COND=(9,LT,PLI),REGION=512K 00380000
//SYSLIB DD DSN=&LIBPRFX..SCEELKED,DISP=SHR               00390000
//SYSPRINT DD SYSOUT=*                                     00400000
//SYSLIN DD DSN=&&LOADSET,DISP=(OLD,DELETE)                 00410000
// DD DDNAME=SYSIN                                         00420000
//SYSLMOD DD DSN=&&GOSET(&GOPGM),DISP=(MOD,PASS),UNIT=SYSDA, 00430000
// SPACE=(1024,(50,20,1))                                  00440000
//SYSUT1 DD DSN=&&SYSUT1,UNIT=SYSDA,SPACE=(1024,(200,20)), 00450000
// DCB=BLKSIZE=1024                                       00460000
//SYSIN DD DUMMY                                           00470000

```

Figure 8. Cataloged Procedure IEL1CL

The linkage editor always places the load modules it creates in the standard data set defined by the DD statement with the name SYSLMOD. This statement in the cataloged procedure specifies a new temporary library &&GOSET, in which the load module will be placed and given the member name GO (unless you specify the NAME compile-time option for the compiler procedure step). In specifying a temporary library, the cataloged procedure assumes that you will run the load module in the same job; if you want to retain the module, you must substitute your own statement for the DD statement with the name SYSLMOD.

The SYSLIN DD statement in Figure 8 shows how to concatenate a data set defined by a DD statement named SYSIN with the primary input (SYSLIN) to the linkage editor. You could place linkage editor control statements in the input stream by this means, as described in the *Language Environment for MVS & VM Programming Guide*.

## Compile, Link-Edit, and Run (IEL1CLG)

This cataloged procedure, shown in Figure 9, includes three procedure steps: PLI, LKED, and GO. PLI and LKED are identical to the two procedure steps of IEL1CL, and GO runs the load module created in the step LKED. The GO step is executed only if no severe or unrecoverable errors occurred in the preceding procedure steps.

Input data for the compilation procedure step should be specified in a DD statement with the name PLI.SYSIN, and for the GO step in a DD statement with the name GO.SYSIN.

---

```

//IEL1CLG PROC LNGPRFX='IEL.V1R1M1',LIBPRFX='CEE.V1R4M0',          00010000
//          SYSLBLK=3200,GOPGM=GO                                00020000
//*                                                    00030000
//***** 00040000
//*                                                    * 00050000
//* LICENSED MATERIALS - PROPERTY OF IBM                    * 00060000
//*                                                    * 00070000
//* 5688-235 (C) COPYRIGHT IBM CORP. 1964, 1995            * 00080000
//* ALL RIGHTS RESERVED                                     * 00090000
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE,           * 00100000
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA            * 00110000
//* ADP SCHEDULE CONTRACT WITH IBM CORP.                   * 00120000
//*                                                    * 00130000
//* SEE COPYRIGHT INSTRUCTIONS                              * 00140000
//*                                                    * 00150000
//***** 00160000
//*                                                    00170000
//* IBM PL/I FOR MVS & VM                                    00180000
//*                                                    00190000
//* COMPILE, LINK EDIT AND RUN A PL/I PROGRAM              00200000
//*                                                    00210000
//* RELEASE LEVEL: 01.01.01 (VERSION.RELEASE.MODIFICATION LEVEL) 00220000
//*                                                    00230000
//* PARAMETER DEFAULT VALUE USAGE                          00240000
//* LNGPRFX IEL.V1R1M1 PREFIX FOR LANGUAGE DATA SET NAMES 00250000
//* LIBPRFX CEE.V1R4M0 PREFIX FOR LIBRARY DATA SET NAMES  00260000
//* SYSLBLK 3200 BLKSIZE FOR OBJECT DATA SET              00270000
//* GOPGM GO MEMBER NAME FOR LOAD MODULE                   00280000
//*                                                    00290000
//PLI EXEC PGM=IEL1AA,PARM='OBJECT,NODECK',REGION=512K      00300000
//STEPLIB DD DSN=&LNGPRFX..SIELCOMP,DISP=SHR                00310000
// DD DSN=&LIBPRFX..SCEERUN,DISP=SHR                        00320000
//SYSPRINT DD SYSOUT=*                                     00330000
//SYSLIN DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,        00340000
// SPACE=(80,(250,100)),DCB=(BLKSIZE=&SYSLBLK)             00350000
//SYSUT1 DD DSN=&&SYSUT1,UNIT=SYSDA,                          00360000
// SPACE=(1024,(200,50)),,CONTIG,ROUND),DCB=BLKSIZE=1024 00370000
//LKED EXEC PGM=IEWL,PARM='XREF,LIST',COND=(9,LT,PLI),REGION=512K 00380000
//SYSLIB DD DSN=&LIBPRFX..SCEELKED,DISP=SHR                 00390000
//SYSPRINT DD SYSOUT=*                                     00400000
//SYSLIN DD DSN=&&LOADSET,DISP=(OLD,DELETE)                  00410000
// DD DDNAME=SYSIN                                         00420000
//SYSLMOD DD DSN=&&GOSSET(&GOPGM),DISP=(MOD,PASS),UNIT=SYSDA, 00430000
// SPACE=(1024,(50,20,1))                                  00440000
//SYSUT1 DD DSN=&&SYSUT1,UNIT=SYSDA,SPACE=(1024,(200,20)),  00450000
// DCB=BLKSIZE=1024                                       00460000
//SYSIN DD DUMMY                                           00470000
//GO EXEC PGM=*.LKED.SYSLMOD,COND=((9,LT,PLI),(9,LT,LKED)), 00480000
// REGION=2048K                                           00490000
//STEPLIB DD DSN=&LIBPRFX..SCEERUN,DISP=SHR                 00500000
//SYSPRINT DD SYSOUT=*                                     00510000
//CEEDUMP DD SYSOUT=*                                     00520000
//SYSUDUMP DD SYSOUT=*                                     00530000

```

---

Figure 9. Cataloged Procedure IEL1CLG

## Compile, Load and Run (IEL1CG)

This cataloged procedure, shown in Figure 10, achieves the same results as IEL1CLG but uses the loader instead of the linkage editor. However, instead of using three procedure steps (compile, link-edit, and run), it has only two (compile and load-and-run). The second procedure step runs the loader program. The loader program processes the object module produced by the compiler and runs the resultant executable program immediately. Input data for the compilation procedure step requires the qualified ddname PLI.SYSIN.

The use of the loader imposes certain restrictions on your PL/I program; before using this cataloged procedure, see *Language Environment for MVS & VM Programming Guide*, which explains how to use the loader.

---

```
// IEL1CG PROC LNGPRFX='IEL.V1R1M1',LIBPRFX='CEE.V1R4M0',          00010000
//          SYSLBLK=3200                                          00020000
//*                                                              00030000
//***** 00040000
//* * 00050000
//* LICENSED MATERIALS - PROPERTY OF IBM * 00060000
//* * 00070000
//* 5688-235 (C) COPYRIGHT IBM CORP. 1964, 1995 * 00080000
//* ALL RIGHTS RESERVED * 00090000
//* US GOVERNMENT USERS RESTRICTED RIGHTS - USE, * 00100000
//* DUPLICATION OR DISCLOSURE RESTRICTED BY GSA * 00110000
//* ADP SCHEDULE CONTRACT WITH IBM CORP. * 00120000
//* * 00130000
//* SEE COPYRIGHT INSTRUCTIONS * 00140000
//* * 00150000
//***** 00160000
//* 00170000
//* IBM PL/I FOR MVS & VM 00180000
//* 00190000
//* COMPILE, LOAD AND RUN A PL/I PROGRAM 00200000
//* 00210000
//* RELEASE LEVEL: 01.01.01 (VERSION.RELEASE.MODIFICATION LEVEL) 00220000
//* 00230000
//* PARAMETER DEFAULT VALUE USAGE 00240000
//* LNGPRFX IEL.V1R1M1 PREFIX FOR LANGUAGE DATA SET NAMES 00250000
//* LIBPRFX CEE.V1R4M0 PREFIX FOR LIBRARY DATA SET NAMES 00260000
//* SYSLBLK 3200 BLKSIZE FOR OBJECT DATA SET 00270000
//* GOPGM GO MEMBER NAME FOR LOAD MODULE 00280000
//* 00290000
//PLI EXEC PGM=IEL1AA,PARM='OBJECT,NODECK',REGION=512K 00300000
//STEPLIB DD DSN=&LNGPRFX..SIELCOMP,DISP=SHR 00310000
// DD DSN=&LIBPRFX..SCEERUN,DISP=SHR 00320000
//SYSPRINT DD SYSOUT=* 00330000
//SYSLIN DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA, 00340000
// SPACE=(80,(250,100)),DCB=(BLKSIZE=&SYSLBLK) 00350000
//SYSUT1 DD DSN=&&SYSUT1,UNIT=SYSDA, 00360000
// SPACE=(1024,(200,50)),,CONTIG,ROUND),DCB=BLKSIZE=1024 00370000
//GO EXEC PGM=LOADER,PARM='MAP,PRINT',COND=(9,LT,PLI), 00380000
// REGION=2048K 00390000
//STEPLIB DD DSN=&LIBPRFX..SCEERUN,DISP=SHR 00400000
//SYSLIB DD DSN=&LIBPRFX..SCEELKED,DISP=SHR 00410000
//SYSPRINT DD SYSOUT=* 00420000
//SYSLIN DD DSN=&&LOADSET,DISP=(OLD,DELETE) 00430000
//SYSLOUT DD SYSOUT=* 00440000
//CEEDUMP DD SYSOUT=* 00450000
//SYSUDUMP DD SYSOUT=* 00460000
```

---

Figure 10. Cataloged Procedure IEL1CG

For more information on other cataloged procedures, see the *Language Environment for MVS & VM Programming Guide*.

---

## Invoking a Cataloged Procedure

To invoke a cataloged procedure, specify its name in the PROC parameter of an EXEC statement. For example, to use the cataloged procedure IEL1C, you could include the following statement in the appropriate position among your other job control statements in the input stream:

```
//stepname EXEC PROC=IEL1C
```

You do not need to code the keyword PROC. If the first operand in the EXEC statement does not begin PGM= or PROC=, the job scheduler interprets it as the name of a cataloged procedure. The following statement is equivalent to that given above:

```
//stepname EXEC IEL1C
```

If you include the parameter MSGLEVEL=1 in your JOB statement, the operating system will include the original EXEC statement in its listing, and will add the statements from the cataloged procedure. In the listing, cataloged procedure statements are identified by XX or X/ as the first two characters; X/ signifies a statement that was modified for the current invocation of the cataloged procedure.

You might be required to modify the statements of a cataloged procedure for the duration of the job step in which it is invoked, either by adding DD statements or by overriding one or more parameters in the EXEC or DD statements. For example, cataloged procedures that invoke the compiler require the addition of a DD statement with the name SYSIN to define the data set containing the source statements. Also, whenever you use more than one standard link-edit procedure step in a job, you must modify all but the first cataloged procedure that you invoke if you want to run more than one of the load modules.

## Specifying Multiple Invocations

You can invoke different cataloged procedures, or invoke the same cataloged procedure several times, in the same job. No special problems are likely to arise unless more than one of these cataloged procedures involves a link-edit procedure step, in which case you must take the following precautions to ensure that all your load modules can be run.

When the linkage editor creates a load module, it places the load module in the standard data set defined by the DD statement with the name SYSLMOD. In the absence of a linkage editor NAME statement (or the NAME compile-time option), it uses the member name specified in the DSNAME parameter as the name of the module. In the standard cataloged procedures, the DD statement with the name SYSLMOD always specifies a temporary library &&GOSET with the member name GO.

If you use the cataloged procedure IEL1CLG twice within the same job to compile, link-edit, and run two PL/I programs, and do not name each of the two load modules that the linkage editor creates, the first load module runs twice, and the second one not at all.

To prevent this, use one of the following methods:

- Delete the library &&GOSET at the end of the GO step. In the first invocation of the cataloged procedure at the end of the GO step, add a DD statement with the syntax:

```
//GO.SYSLMOD DD DSN=&&GOSET,  
//  DISP=(OLD,DELETE)
```

- Modify the DD statement with the name SYSLMOD in the second and subsequent invocations of the cataloged procedure so as to vary the names of the load modules.

For example:

```
//LKED.SYSLMOD DD DSN=&&GOSET(G01)
```

and so on.

- Use the NAME compile-time option to give a different name to each load module and change each job step EXEC statement to specify the running of the load module with the name for that job step.
- Use the NAME linkage editor option to give a different name to each load module and change each job step EXEC statement to specify the running of the load module with the name for that job step.

**Note:** To assign a membername to the load module, you can use either the compile-time or linkage editor NAME option with the DSNAME parameter on the SYSLMOD DD statement. When you use this procedure, the membername *must* be identical to the name on the NAME option if the EXEC statement that runs the program refers to the SYSLMOD DD statement for the name of the module to be run.

Another option is to give each program a different name by using GOPGM on the EXEC procedure statement. For example:

```
// EXEC IEL1CLG,GOPGM=G02
```

## Link-Editing Multitasking Programs

When you use a cataloged procedure to link-edit a multitasking program, the load module must include the multitasking versions of the PL/I library subroutines.

To ensure that the multitasking library (SYS1.SIBMTASK) is searched before the base library, include the parameter LKLBDSN='SYS1.SIBMTASK' in the EXEC statement that invokes the cataloged procedure.

For example:

```
//STEPA EXEC IEL1CLG,LKLBDSN='SYS1.PLITASK'
```

In the standard cataloged procedures the DD statement SYSLIB is always followed by another, unnamed, DD statement that includes the parameter DSNAME=SYS1.SCEELKED. The effect of this statement is to concatenate the base library with the multitasking library. When LKLBDSN=SYS1.SIBMASE is specified, the second DD statement has no effect.

---

## Modifying the PL/I Cataloged Procedures

You can modify a cataloged procedure temporarily by including parameters in the EXEC statement that invokes the cataloged procedure, or by placing additional DD statements after the EXEC statement. Temporary modifications apply only for the duration of the job step in which the procedure is invoked. They do not affect the master copy of the cataloged procedure in the procedure library.

Temporary modifications can apply to EXEC or DD statements in a cataloged procedure. To change a parameter of an EXEC statement, you must include a corresponding parameter in the EXEC statement that invokes the cataloged procedure. To change one or more parameters of a DD statement, you must include a corresponding DD statement after the EXEC statement that invokes the cataloged procedure. Although you cannot add a new EXEC statement to a cataloged procedure, you can always include additional DD statements.

### EXEC Statement

If a parameter of an EXEC statement that invokes a cataloged procedure has an unqualified name, the parameter applies to all the EXEC statements in the cataloged procedure. The effect on the cataloged procedure depends on the parameters, as follows:

- PARM applies to the first procedure step and nullifies any other PARM parameters.
- COND and ACCT apply to all the procedure steps.
- TIME and REGION apply to all the procedure steps and override existing values.

For example, the statement:

```
//stepname EXEC IEL1CLG,PARM='SIZE(MAX)',REGION=512K
```

- Invokes the cataloged procedure IEL1CLG.
- Substitutes the option SIZE(MAX) for OBJECT and NODECK in the EXEC statement for procedure step PLI.
- Nullifies the PARM parameter in the EXEC statement for procedure step LKED.
- Specifies a region size of 512K for all three procedure steps.

To change the value of a parameter in only one EXEC statement of a cataloged procedure, or to add a new parameter to one EXEC statement, you must identify the EXEC statement by qualifying the name of the parameter with the name of the procedure step. For example, to alter the region size for procedure step PLI only in the preceding example, code:

```
//stepname EXEC PROC=IEL1CLG,PARM='SIZE(MAX)',REGION.PLI=512K
```

A new parameter specified in the invoking EXEC statement overrides completely the corresponding parameter in the procedure EXEC statement.

You can nullify all the options specified by a parameter by coding the keyword and equal sign without a value. For example, to suppress the bulk of the linkage editor listing when invoking the cataloged procedure IEL1CLG, code:

```
//stepname EXEC IEL1CLG,PARM.LKED=
```

## DD Statement

To add a DD statement to a cataloged procedure, or to modify one or more parameters of an existing DD statement, you must include a DD statement with the form "procstepname.ddname" in the appropriate position in the input stream. If "ddname" is the name of a DD statement already present in the procedure step identified by "procstepname," the parameters in the new DD statement override the corresponding parameters in the existing DD statement; otherwise, the new DD statement is added to the procedure step. For example, the statement:

```
//PLI.SYSIN DD *
```

adds a DD statement to the procedure step PLI of cataloged procedure IEL1C and the effect of the statement:

```
//PLI.SYSPRINT DD SYSOUT=C
```

is to modify the existing DD statement SYSPRINT (causing the compiler listing to be transmitted to the system output device of class C).

Overriding DD statements must appear after the procedure invocation and in the same order as they appear in the cataloged procedure. Additional DD statements can appear after the overriding DD statements are specified for that step.

To override a parameter of a DD statement, code either a revised form of the parameter or a replacement parameter that performs a similar function (for example, SPLIT for SPACE). To nullify a parameter, code the keyword and equal sign without a value. You can override DCB subparameters by coding only those you wish to modify; that is, the DCB parameter in an overriding DD statement does not necessarily override the entire DCB parameter of the corresponding statement in the cataloged procedures.

---

## Chapter 3. Compiling under MVS

This chapter describes how to invoke the compiler under TSO and the job control statements used for compiling under MVS. You must be linked to Language Environment before you can compile your program.

---

### Invoking the Compiler under TSO

The usual method of invoking the compiler is with the PLI command. In its simplest form the command consists of the keyword and the name of the TSO data set holding the PL/I source program. For example:

```
PLI CALTROP
```

In addition to the data set name, you can specify the PRINT operand to control the compiler listings, and the LIB operand to specify secondary input data sets for the %INCLUDE statements. You can also specify compile-time options as operands of the PLI command.

The command processor for the PLI command is a program known as the *PL/I prompter*. When you enter the command, this program checks the operands and allocates the data sets required by the compiler. Then, it passes control to the compiler and displays a message.

If the source data set has a conventional TSO data set name, you can use the simple name, as in the example above. If not, you need to specify the full name and enclose it in single quotation marks:

```
PLI 'DIANTHUS'
```

or

```
PLI 'JJONES.ERICA.PLI'
```

The compiler translates the source program into object modules, which it stores on external data sets. You can link-edit and run these object modules conversationally.

If you use an unqualified data set name, as in the example at the start of this section, the system generates a name for the object module data set. It takes the simple name of the source data set—CALTROP in the example—and adds your user-identification and the descriptive qualifier OBJ. Hence, if the user who entered the example PLI command had the identification WSMITH, the object module would be written onto a data set called WSMITH.CALTROP.OBJ.

You can make your own choice of name for the object module data set by including the OBJECT compile-time option as an operand of the PLI command. For example:

```
PLI CALTROP OBJECT(TRAPA)
```

The system adds the same qualifiers to this name as it does to the source data set simple name, so the object module is written onto a data set, in this example, called WSMITH.TRAPA.OBJ.

You can specify the full name of the object module data set by enclosing it in quotation marks. For example:

```
PLI CALTROP OBJECT('NATANS')
```

The system in this case adds no qualifiers, so the object module is stored on a data set called NATANS.

You can specify a full name to store the object module with another user's user-identification. For instance, the following command would store the object module using the user-identification JJONES:

```
PLI CALTROP OBJECT('JJONES.CALTROP.OBJ')
```

An alternative to the PLI command is the RUN command or subcommand.

## Allocating Data Sets

The compiler requires the use of a number of data sets in order to process a PL/I program. These are listed in Table 10 on page 58. The following data sets are always required by the compiler:

- The data set holding the PL/I program
- A data set for the compiler listing.

Up to six data sets, including the above two, can be required, depending on which compile-time options have been specified.

These data sets must be allocated before the compiler can use them. If you use the PLI command or the RUN command or subcommand, you invoke the compiler via the prompter, and the prompter allocates the necessary data sets. If you invoke the compiler without the prompter, you must allocate the necessary data sets yourself.

When the prompter allocates compiler data sets, it uses ddnames generated by TSO rather than the ddnames that are used in batch mode. Table 10 on page 58 includes the batch-mode ddnames of the data sets. If the compiler is invoked via the prompter, you cannot refer to the data sets by these names. To control the allocation of compiler data sets, you need to use the appropriate operand of the PLI command. For instance, to allocate the standard output file (ddname SYSPRINT in batch mode) to the terminal, you should use the PRINT(\*) operand of the PLI command. You cannot make the allocation by using the ALLOCATE command with FILE(SYSPRINT) and DATASET(\*) operands. Table 10 on page 58 shows which operands to use for those data sets whose allocation you can control.

When the prompter is not invoked, the batch-mode ddnames are recognized as referring to the compiler data sets.

Table 10. Compiler Data Sets

Data set (and batch-mode ddname)	When required	Where to specify data set in PLI command	Descriptive qualifier	Allocated by	Parameters used by prompter <sup>1</sup> SPACE= <sup>2</sup>	Parameters used by prompter <sup>1</sup> DISP= <sup>3</sup>
Primary input (SYSCIN or SYSIN)	Always	1st operand	PLI	Prompter	— <sup>4</sup>	SHR
Temporary work data set (SYSUT1)	When large program spills internal text pages	Cannot specify	—	Prompter	(1024,(60,60))	(NEW,DELETE)
Compiler listing (SYSPRINT)	Always	Argument of PRINT operand	LIST	Prompter	(629,(n,m))	(OLD,KEEP) or <sup>5</sup> (NEW,CATLG)
Object module (SYSLIN)	When OBJECT option applies	1st argument of OBJECT operand	OBJ	Prompter, when required <sup>6</sup>	(400,(50,50))	(OLD,KEEP) or (NEW,CATLG)
Object module or preprocessor output in card format (SYSPUNCH)	When either DECK or MACRO and MDECK options apply	Argument of MDECK DECK operand	DECK or MACRO and MDECK	Prompter, when required <sup>6</sup>	(400,(50,50))	(OLD,KEEP) or (NEW,CATLG)
Secondary input to preprocessor (SYSLIB) <sup>7</sup>	When &INCLUDE files are used	Arguments of LIB operand	INCLUDE or MACRO	Prompter, when required	— <sup>7</sup>	SHR

**Notes:**

- Unit is determined by entry in User Attribute Data Set.
- These space allocations apply only if the data set is new. The first argument of the SPACE parameter establishes the block size. For the SYSUT1, SYSPRINT, SYSLIN, and SYSPUNCH data sets, the record format, record length, and number of buffers are established by the compiler when it opens the data sets.
- The prompter first tries to allocate the SYSPRINT, SYSLIN, and SYSPUNCH data sets with DISP=(OLD,KEEP). This will cause any existing data set (or partitioned data set member) with the same name to be replaced with the new one. If the data set name cannot be found in the system catalog, the data set is allocated with DISP=(NEW,CATLG).
- The data set already exists; therefore, SPACE (and also UNI T) are already established.
- DISP parameter used only if PRINT(dsname) operand applies. Otherwise, prompter supplies the following parameters:  
 TERM=TS if PRINT(\*) operand applies  
 DUMMY if NOPRINT operand applies  
 SYSOUT if SYSPRINT operand applies.
- Except when the associated option has been specified by means of a %PROCESS statement. In this case, the data set(s) must be allocated by the user.
- If any ddnames are specified in %INCLUDE statements, allocate the data sets with the ALLOCATE statement.

## Using the PLI Command

Use the PLI command to compile a PL/I program. The command invokes the PL/I prompter to process the operands and call the compiler, according to the syntax shown in the following table:

Table 11. Syntax of the PLI Command

COMMAND	OPERANDS
PLI	data-set-name [option-list] PRINT[(*) { (dsname[, [n] [,m]])] SYSPRINT[(sysout-class[, [n] [,m]])] NOPRINT [LIB(dslist)]

### data-set-name

specifies the name of the primary input data set for the compiler. This can be either a fully qualified name (enclosed in single quotation marks) or a simple name (for which the prompter adds the identification qualifier, and the descriptive qualifier PLI). This must be the first operand specified.

### option-list

specifies one or more compile-time options that apply for this compilation.

The compile-time options that you can specify in a TSO environment are described later in this section. Programmers familiar with batch processing should note that defaults are altered for TSO, and that the DECK, MDECK, and OBJECT options are extended to allow specific names of data sets onto which the output is written.

Separate the options by at least one blank or one comma; you can add any number of extra blanks. The order of the options is unimportant. In fact, the PRINT/NOPRINT and LIB operands can be interspersed in the option-list since they are recognized by their keywords. If two contradictory options are specified, the last is accepted and the first ignored.

Options specified in the PLI command can be overridden by options specified on the %PROCESS compiler control statements in the primary input. If the DECK, MDECK, and OBJECT options are required for any program in a batched compilation, the option should be specified in the PLI command so that the prompter allocates the required data sets. The negative forms can then be used on the %PROCESS statements for the programs that do not require the option. The options are described below.

**DECK[(dsname)]:** This can be a fully qualified name (enclosed in single quotation marks) or a simple name (to which the user identification and descriptive qualifier DECK is added). If *dsname* is not specified, the user-supplied name is taken from the first operand of the PLI command, and the user-identification and descriptive qualifier DECK is added. If *dsname* is not specified and the first operand of the PL/I command specifies a member of a partitioned data set, the member name is ignored—the generated

data set name is based on the name of the partitioned data set. For more information on this option see DECK on page 10.

**MDECK[(dsname)]:** This can be a fully qualified name (enclosed in single quotation marks) or a simple name (to which the user identification and descriptive qualifier MDECK is added). If *dsname* is not specified, the user-supplied name is taken from the first operand of the PLI command, and the user-identification and descriptive qualifier MDECK are added. If *dsname* is not specified and the first operand of the PLI command specifies a member of a partitioned data set, the member name is ignored—the generated data set name is based on the name of the partitioned data set. For more information on this option, see MDECK on page 17.

**OBJECT [(dsname)]:** This can be a fully qualified name (enclosed in single quotation marks) or a simple name (to which the user identification and the descriptive qualifier OBJ is added). If *dsname* is not specified, the user-supplied name is taken from the first operand of the PLI command, and the user-identification and descriptive qualifier OBJ are added. If *dsname* is not specified and the first operand of the PLI command specifies a member of a partitioned data set, the member name is ignored—the generated data set name is based on the name of the partitioned data set. For more information on this option, see OBJECT on page 19.

**PRINT(\*)** specifies that the compiler listing, on the SYSPRINT file, is written at the terminal; no other copy will be available. The PRINT(\*) operand is implemented by generating a TERMINAL option with a list of options which correspond to the listings printed at the terminal. If you specify the TERMINAL option after the PRINT(\*) operand, this overrides the TERMINAL option generated by the PRINT(\*) operand.

**PRINT(dsname[,n][,m])**

specifies that the compiler listing, on the SYSPRINT file, is written on the data set named in parentheses. This can be either a fully qualified name (enclosed in single quotation marks) or a simple name (for which the prompter adds the identification qualifier, and the description qualifier LIST).

If you do not specify a *dsname* argument for the PRINT operand, the prompter adds the identification and descriptive qualifiers to the data set name specified in the first operand, producing a data set name of the form:

user-identification.user-supplied-name.LIST

If *dsname* is not specified and the first operand of PLI specifies a member of a partitioned data set, the member name is ignored—the generated data set name is based on the name of the partitioned data set.

In this command, *n* and *m* specify the space allocation in lines for the listing data set. They should be used when the size of the listing has caused a B37 abend during compilation.

*n* specifies the number of lines in the primary allocation.

*m* specifies the number of lines in the secondary allocation.

If *n* is omitted, the preceding comma must be included. For example, to enter only the size of the secondary allocation and accept the default for the primary, you would enter:

```
PRINT(printds,,500)
```

The space allocation used if *n* and *m* are not specified is the allocation specified during compiler installation.

**SYSPRINT [(sysout-class[,*n*][,*m*])]**

specifies that the compiler listing, on the SYSPRINT file, is to be written to the sysout class named in parentheses. If no class is specified, the output is written to a default sysout class. The IBM-supplied standard for this default is class A. For an explanation of the *n* and *m* see the "PRINT" operand above.

**NOPRINT**

specifies that the compiler listing is not produced on the SYSPRINT file. You can still get most of the listing written at the terminal by using the TERMINAL compile-time option.

**LIB(dslist)**

specifies one or more data sets that are used as the secondary input to the preprocessor. These data sets are concatenated in the order specified and then associated with the ddname in the %INCLUDE statement in the PL/I program. You must allocate the data sets associated with that ddname yourself.

The data set names can be either fully qualified (each enclosed in single quotation marks) or simple names (for which the prompter adds the identification qualifier, but no descriptive qualifier).

Separate the data set names by at least one blank or one comma; you can add any number of extra blanks.

If you use the LIB operand, either the INCLUDE or the MACRO compile-time option must also apply.

The following examples give an operation to be performed and the known variables, and show you how to enter the command to perform that particular function.

**Example 1**

Operation: Invoke the compiler to process a PL/I program.

Known: – User-identification is ABC.

– Data set containing the program is named ABC.UPDATE.PLI.

– SYSPRINT file is to be directed to the terminal.

– Default options and data set names are to be used.

Command: PLI UPDATE PRINT(\*)

## Example 2

Operation: Invoke the compiler to process a PL/I program.

Known: – User-identification is XYZ.

– Data set containing the program is named  
ABC.MATRIX.PLI.

– SYSPRINT file is to be written on a data set named  
MATLIST.

– MACRO and MDECK options are required, with the  
associated output to be written on a data set named  
MATCARD.

– Secondary input to preprocessor to be read from  
a library named XYZ.SOURCE.

– Otherwise default options and data set names  
are to be used.

Command: `PLI 'ABC.MATRIX.PLI' +  
PRINT('MATLIST'),MACRO,MDECK('MATCARD'), +  
LIB(SOURCE)`

## Compiler Listings

In conversational mode, as in batch mode, compile-time options control which listings the compiler produces (see Chapter 1, “Using Compile-Time Options and Facilities” on page 5). You can specify the options as operands of the PLI command.

In addition to specifying which listings are to be produced, you need to indicate where they are to be transmitted. If you wish to have them displayed at the terminal, you can specify either the PRINT(\*) operand, which allocates the compiler listing file to the terminal, or the TERMINAL option. The latter should contain a list of the options corresponding to the listings you require at the terminal. For instance, to produce a source listing at the terminal, you could enter either:

```
PLI CALTROP PRINT(*) SOURCE
```

or:

```
PLI CALTROP TERM(SOURCE)
```

Compiler listings can be directed to a data set by specifying the PRINT operand with the data set's name, or to a SYSOUT class by specifying the SYSPRINT operand. For further details see “Using the Compiler Listing” on page 33 and “Listing (SYSPRINT)” on page 67.

### Using %INCLUDE under TSO

In conversational mode, as in batch mode, you can incorporate PL/I source code into your program by means of the %INCLUDE statement. This statement names members of partitioned data sets that hold the code to be included. You can create these secondary input data sets either under TSO or in batch mode.

To use %INCLUDE you must specify the MACRO or INCLUDE compile-time option.

The %INCLUDE statement can specify simply the name of the data set member that holds the text to be included. For instance:

```
%INCLUDE RECDCL;
```

It can also specify a ddname that is associated with the member. For example:

```
%INCLUDE STDCL (F726);
```

STDCL is the ddname, and F726 is the member name. A single %INCLUDE statement can specify several data set members, and can contain both forms of specification. For example:

```
%INCLUDE SUBA(READ5),SUBC(REPORT1),DATEFUNC;
```

### Allocating Data Sets in %INCLUDE

All data sets containing secondary input must be allocated before the compiler is invoked. If a data set member is specified in an %INCLUDE statement without a ddname, the data set can be allocated by specifying the data set name in the LIB operand of the PLI command. (This operand is the equivalent of the batch-mode SYSLIB DD statement.) The necessary allocation is made by the PL/I prompter.

If a ddname has been specified in the %INCLUDE statement, the corresponding data set must be allocated by means of either an ALLOCATE command or the logon procedure.

Suppose the data set members specified in the %INCLUDE statements in the preceding section are held on data sets as follows (the ddname used in the %INCLUDE statement is also shown):

Member:	Data Set Name:	DDNAME:
RECDCL	LDSRCE	none
F726	WPSRCE	STDCL
READ5	JESRCE	SUBA
REPORT	GHSRCE	SUBC
DATEFUNC	DRSRCE	none

Then the necessary data sets could be allocated by the following commands:

```
ALLOCATE FILE(STDCL) DATASET(WPSRCE)
ALLOCATE FILE(SUBA) DATASET(JESRCE)
ALLOCATE FILE(SUBC) DATASET(GHSRCE)
PLI MNTHCOST LIB(LDSRCE,DRSRCE) INCLUDE
```

## Running Jobs in a Background Region

If you have the necessary authorization, you can submit jobs for processing in a background region. Your installation must record the authorization in your UADS (User Attribute Data Set) entry.

Jobs are submitted by means of the SUBMIT command. The command must include the name of the data set holding the job or jobs to be processed, and the data set must contain the necessary Job Control Language statements. Jobs will run under the same version of the operating system as is used for TSO. Output from the jobs can be manipulated from your terminal.

Further details about submitting background jobs are given in the manual *TSO Terminal User's Guide*.

---

## Using JCL during Compilation

Although you will probably use cataloged procedures rather than supply all the job control required for a job step that invokes the compiler, you should be familiar with these statements so that you can make the best use of the compiler and, if necessary, override the statements of the cataloged procedures.

The IBM-supplied cataloged procedures that include a compilation procedure step are:

<b>IEL1C</b>	Compile only
<b>IEL1CL</b>	Compile and link-edit
<b>IEL1CLG</b>	Compile, link-edit, and run
<b>IEL1CG</b>	Compile, load and run

The following paragraphs describe the job control statements needed for compilation. The IBM-supplied cataloged procedures described in "IBM-Supplied Cataloged Procedures" on page 46 contain these statements. Therefore, you need to code them yourself only if you are not using the cataloged procedures.

## EXEC Statement

The basic EXEC statement is:

```
//stepname EXEC PGM=IEL1AA
```

512K is required for the REGION parameter of this statement. The PARM parameter of the EXEC statement can be used to specify one or more of the optional facilities provided by the compiler. These facilities are described under "Specifying Options in the EXEC Statement" on page 68. See Chapter 1, "Using Compile-Time Options and Facilities" on page 5 for a description of the options.

## DD Statements for the Standard Data Sets

The compiler requires several standard data sets, the number of data sets depends on the optional facilities specified. You must define these data sets in DD statements with the standard ddnames which are shown, together with other characteristics of the data sets, in Table 12 on page 65. The DD statements SYSIN, SYSUT1, and SYSPRINT are always required.

You can store any of the standard data sets on a direct-access device, but you must include the SPACE parameter in the DD statement. This parameter defines the data set to specify the amount of auxiliary storage required. The amount of auxiliary storage allocated in the IBM-supplied cataloged procedures should suffice for most applications.

Table 12. Compiler Standard Data Sets

Standard DDNAME	Contents of data set	Possible device classes <sup>1</sup>	Record format (RECFM) <sup>2</sup>	Record size (LRECL) <sup>3</sup>	BLKSIZE
SYSIN (or SYSCIN) <sup>4</sup>	Input to the compiler	SYSSQ	<i>F,FB,U</i> <i>VB,V</i>	<101(100) <105(104)	—
SYSLIN	Object module	SYSSQ	FB	80	80
SYSPUNCH	Preprocessor output, compiler output	SYSSQ SYSCP	FB	80	80
SYSUT1	Temporary workfile	SYSDA	<i>F</i>	4051	—
SYSPRINT	Listing, including messages	SYSSQ	<i>VBA</i>	125	129
SYSLIB	Source statements for preprocessor	SYSDA	<i>F,FB,U</i> <i>V,VB</i>	<101 <105	—

**Notes:**

The only value for compile-time SYSPRINT that can be overridden is BLKSIZE.

1. The possible device classes are:

- SYSSQ Sequential device
- SYSDA Direct-access device
- SYSCP Card-punch device.

Block size can be specified except for SYSUT1. The block size and logical record length for SYSUT1 is chosen by the compiler.

- 2. If the record format is not specified in a DD statement, the default value is provided by the compiler. (Default values are shown in italics.)
- 3. The numbers in parentheses in the "Record Size" column are the defaults, which you can override.
- 4. The compiler will attempt to obtain source input from SYSCIN if a DD statement for this data set is provided. Otherwise it will obtain its input from SYSIN.

### Input (SYSIN or SYSCIN)

Input to the compiler must be a data set defined by a DD statement with the name SYSIN or SYSCIN. This data set must have CONSECUTIVE organization. The input must be one or more external PL/I procedures. If you want to compile more than one external procedure in a single job or job step, precede each procedure, except possibly the first, with a %PROCESS statement. For further detail, see "Compiling Multiple Procedures in a Single Job Step" on page 69.

80-byte records are commonly used as the input medium for PL/I source programs. The input data set can be on a direct-access device, magnetic tape, or some other sequential media. The input data set can contain either fixed-length records (blocked or unblocked), variable-length records (coded or uncoded), or undefined-length records. The maximum record size is 100 bytes.

When data sets are concatenated for input to the compiler, the concatenated data sets must have similar characteristics (for example, block size and record format).

### Output (SYSLIN, SYSPUNCH)

Output in the form of one or more object modules from the compiler can be stored in either of two data sets. You can store it in the data set SYSLIN (if you specify the OBJECT compile-time option) or in the data set SYSPUNCH (if you specify the DECK compile-time option). Both of these data sets are defined by the DD statement. You can specify both the OBJECT and DECK options in one program, if the output will be stored in both data sets.

The object module is always in the form of 80-byte fixed-length records, blocked or unblocked. The data set defined by the DD statement with the name SYSPUNCH is also used to store the output from the preprocessor if you specify the MDECK compile-time option.

## Temporary Workfile (SYSUT1)

The compiler requires a data set for use as a temporary workfile. It is defined by a DD statement with the name SYSUT1, and is known as the *spill file*. It must be on a direct-access device, and must not be allocated as a multi-volume data set.

The spill file is used as a logical extension to main storage and is used by the compiler and by the preprocessor to contain text and dictionary information. The LRECL and BLKSIZE for SYSUT1 is chosen by the compiler based on the amount of storage available for spill file pages.

The DD statements given in this publication and in the cataloged procedures for SYSUT1 request a space allocation in blocks of 1024 bytes. This is to insure that adequate secondary allocations of direct-access storage space are acquired.

### Statement Lengths

The compiler has a restriction that any statement must fit into the compiler's work area. The maximum size of this work area varies with the amount of space available to the compiler. The maximum length of a statement is 3400 characters.

The DECLARE statement is an exception in that it can be regarded as a sequence of separate statements, each of which starts wherever a comma occurs that is not contained within parentheses. For example:

```
DCL 1 A,  
    2 B(10,10) INIT(1,2,3,...),  
    2 C(10,100) INIT((1000)(0)),  
    (D,E) CHAR(20) VAR,...
```

In this example, each line can be treated by the compiler as a separate DECLARE statement in order to accommodate it in the work area. The compiler will also treat the INITIAL attribute in the same way when it is followed by a list of items separated by commas that are not contained within parentheses. Each item can contain initial values that, when expanded, do not exceed the maximum length. The above also applies to the use of the INITIAL attribute in a DEFAULT statement.

If a DECLARE statement cannot be compiled, the following techniques are suggested to overcome this problem:

- Simplify the DECLARE statement so that the compiler can treat the statement in the manner described above.

- Modify any lists of items following the INITIAL attribute so that individual items are smaller and separated by commas not contained in parentheses. For example, the following declaration is followed by an expanded form of the same declaration. The compiler can more readily accommodate the second declaration in its work area:

1. DCL Y (1000) CHAR(8)  
INIT ((1000) (8)'Y');
2. DCL Y (1000) CHAR(8) INIT  
((250) (8)'Y', (250) (8)'Y',  
(250) (8)'Y', (250) (8)'Y');

## Listing (SYSPRINT)

The compiler generates a listing that includes all the source statements that it processed, information relating to the object module, and, when necessary, messages. Most of the information included in the listing is optional, and you can specify those parts that you require by including the appropriate compile-time options. The information that can appear, and the associated compile-time options, are described under “Using the Compiler Listing” on page 33.

You must define the data set, in which you wish the compiler to store its listing, in a DD statement with the name SYSPRINT. This data set must have CONSECUTIVE organization. Although the listing is usually printed, it can be stored on any sequential or direct-access device. For printed output, the following statement will suffice if your installation follows the convention that output class A refers to a printer:

```
//SYSPRINT DD SYSOUT=A
```

The compiler always reserves 258 bytes of main storage (129 bytes each) for two buffers for this data set. However, you can specify a block size of more than 129 bytes, provided that sufficient main storage is available to the compiler. (For further details of the SIZE compile-time option, see SIZE on page 22.)

## Source Statement Library (SYSLIB)

If you use the preprocessor %INCLUDE statement to introduce source statements into the PL/I program from a library, you can either define the library in a DD statement with the name SYSLIB, or you can choose your own ddname (or ddnames) and specify a ddname in each %INCLUDE statement. (For further information on the preprocessor, see “Using the Preprocessor” on page 29.)

If the statements are included from a SYSLIB, they must have a form that is similar to the %INCLUDE statement. For example, they must have the same record format (fixed, variable, undefined), the same logical record length, and matching left and right margins.

The BLOCKSIZE of the library must be less than or equal to 32,760 bytes.

## Example of Compiler JCL

A typical sequence of job control statements for compiling a PL/I program is shown in Figure 11 on page 68. The DECK and NOOBJECT compile-time options, described below, have been specified to obtain an object module as a card deck only.

---

```

//OPT4#4 JOB
//STEP EXEC PGM=IEL1AA,PARM='DECK,NOOBJECT'
//STEPLIB DD DSN=IEL.V1R1M1.SIELCOMP,DISP=SHR
// DD DSN=CEE.V1R2M0.SCEERUN,DISP=SHR
//SYSPUNCH DD SYSOUT=B
//SYSUT1 DD UNIT=SYSDA,SPACE=(1024,(60,60),,CONTIG)
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
/*

```

---

Figure 11. Job Control Statements for Compiling a PL/I Program Not Using Cataloged Procedures

## Specifying Options

For each compilation, the IBM-supplied or installation default for a compile-time option applies unless it is overridden by specifying the option in a %PROCESS statement or in the PARM parameter of an EXEC statement.

An option specified in the PARM parameter overrides the default value, and an option specified in a %PROCESS statement overrides both that specified in the PARM parameter and the default value.

**Note:** When conflicting attributes are specified either explicitly or implicitly by the specification of other options, the latest implied or explicit option is accepted. No diagnostic message is issued to indicate that any options are overridden in this way.

## Specifying Options in the EXEC Statement

To specify options in the EXEC statement, code PARM= followed by the list of options, in any order (except that CONTROL, if used, must be first) separating the options with commas and enclosing the list within single quotation marks, for example:

```
//STEP1 EXEC PGM=IEL1AA,PARM='OBJECT,LIST'
```

Any option that has quotation marks, for example MARGINI('c'), must have the quotation marks duplicated. The length of the option list must not exceed 100 characters, including the separating commas. However, many of the options have an abbreviated syntax that you can use to save space. If you need to continue the statement onto another line, you must enclose the list of options in parentheses (instead of in quotation marks) enclose the options list on each line in quotation marks, and ensure that the last comma on each line except the last line is outside of the quotation marks. An example covering all the above points is as follows:

```
//STEP1 EXEC PGM=IEL1AA,PARM=('AG,A',
//      'C,ESD,F(I)',
//      'M,MI(''X''),NEST,STG,X')
```

If you are using a cataloged procedure, and want to specify options explicitly, you must include the PARM parameter in the EXEC statement that invokes it, qualifying the keyword PARM with the name of the procedure step that invokes the compiler. For example:

```
//STEP1 EXEC IEL1CLG,PARM.PLI='A,LIST,ESD'
```

## Compiling Multiple Procedures in a Single Job Step

Batched compilation allows the compiler to compile more than one external PL/I procedure in a single job step. The compiler creates an object module for each external procedure and stores it sequentially either in the data set defined by the DD statement with the name SYSPUNCH, or in the data set defined by the DD statement with the name SYSLIN. Batched compilation can increase compiler throughput by reducing operating system and compiler initialization overheads.

To specify batched compilation, include a compiler %PROCESS statement as the first statement of each external procedure except possibly the first. The %PROCESS statements identify the start of each external procedure and allow compile-time options to be specified individually for each compilation. The first procedure might require a %PROCESS statement of its own, because the options in the PARM parameter of the EXEC statement apply to all procedures in the batch, and can conflict with the requirements of subsequent procedures.

**Note:** The options specified in the %PROCESS statement override those specified in the PARM parameter of the EXEC statement.

The method of coding a %PROCESS statement and the options that can be included are described under “Specifying Options in the %PROCESS or \*PROCESS statements” on page 28. The options specified in a %PROCESS statement apply to the compilation of the source statements between that %PROCESS statement and the next %PROCESS statement. Options other than these, either the defaults or those specified in the PARM field, will also apply to the compilation of these source statements. Two options, the SIZE option and the NAME option have a particular significance in batched compilations, and are discussed below.

**Note:** OBJECT, MDECK, and DECK can cause problems if they are specified on second or subsequent compilations but not on the first. This is because they require the opening of SYSLIN or SYSPUNCH and there might not be room for the associated data management routines and control blocks. When this happens, compilation ends with a storage abend.

### SIZE Option

In a batched compilation, the SIZE specified in the first procedure of a batch (by a %PROCESS or EXEC statement, or by default) is used throughout. If SIZE is specified in subsequent procedures of the batch, it is diagnosed and ignored.

### NAME Option

The NAME option specifies that the compiler places a linkage editor NAME statement as the last statement of the object module. The use of this option in the PARM parameter of the EXEC statement, or in a %PROCESS statement, determines how the object modules produced by a batched compilation are handled by the linkage editor. When the batch of object modules is link-edited, the linkage editor combines all the object modules between one NAME statement and the preceding NAME statement into a single load module. It takes the name of the load module from the NAME statement that follows the last object module that is included. When combining two object modules into one load module, the NAME option should not be used in the EXEC statement. An example of the use of the NAME option is given in Figure 12 on page 70.

---

```

// EXEC IEL1C,PARM.PLI='LIST'
.
.
.
% PROCESS NAME('A');
ALPHA: PROC OPTIONS(MAIN);
.
.
.
END ALPHA;
% PROCESS;
BETA: PROC;
.
.
.
END BETA;
% PROCESS NAME('B');
GAMMA: PROC;
.
.
.
END GAMMA;

```

---

*Figure 12. Use of the NAME Option in Batched Compilation*

Compilation of the PL/I procedures ALPHA, BETA, and GAMMA, results in the following object modules and NAME statements:

```

OBJECT MODULE FOR ALPHA
NAME A (R)
OBJECT MODULE FOR BETA
OBJECT MODULE FOR GAMMA
NAME B (R)

```

From this sequence of object modules and control statements, the linkage editor produces two load modules, one named A containing the object module for the external PL/I procedure ALPHA, and the other named B containing the object modules for the external PL/I procedures BETA and GAMMA.

**Note:** You should not specify the option NAME if you intend to process the object modules with the loader. The loader processes all object modules into a single load module. If there is more than one name, the loader recognizes the first one only and ignores the others.

## Return Codes in Batched Compilation

The return code generated by a batched compilation is the highest code that is returned if the procedures are compiled separately.

## Job Control Language for Batched Processing

The only special consideration relating to JCL for batched processing refers to the data set defined by the DD statement with the name SYSLIN. If you include the option OBJECT, ensure that this DD statement contains the parameter DISP=(MOD,KEEP) or DISP=(MOD,PASS). (The IBM-supplied cataloged procedures specify DISP=(MOD,PASS).) If you do not specify DISP=MOD, successive object modules will overwrite the preceding modules.

## Examples of Batched Compilations

If the external procedures are components of a large program and need to be run together, you can link-edit them together and run them in subsequent job steps. Cataloged procedure IEL1CG can be used, as shown in Figure 13.

---

```
//OPT4#13 JOB
//STEP1 EXEC IEL1CG
//PLI.SYSIN DD *
    First PL/I source program
% PROCESS;
    Second PL/I source program
% PROCESS;
    Third PL/I source program
/*
//GO.SYSIN DD *
    Data processed by combined
    PL/I programs

/*
```

---

Figure 13. Example of Batched Compilation, Including Execution

If the external procedures are independent programs to be invoked individually from a load module library, cataloged procedure IEL1CL can be used. For example, a job that contains three compile and link-edit operations can be run as a single batched compilation, as shown in Figure 14.

---

```
//OPT4#14 JOB
//STEP1 EXEC IEL1CL,
// PARM.PLI='NAME(''PROG1'')',
// PARM.LKED=LIST
//PLI.SYSIN DD *
    First PL/I source program
% PROCESS NAME('PROG2');
    Second PL/I source program
% PROCESS NAME('PROG3');
    Third PL/I source program
/*
//LKED.SYSLMOD DD DSN=PUBPGM,
// DISP=OLD
```

---

Figure 14. Example of Batched Compilation, Excluding Execution

---

## Correcting Compiler-Detected Errors

At compile time, both the preprocessor and the compiler can produce diagnostic messages and listings. For information on correcting errors, see “Correcting Compiler-Detected Errors” on page 78 in Chapter 4, “Compiling under VM.”

---

## The PL/I Compiler and MVS/ESA

Care should be taken when using large region sizes with the SIZE(MAX) compiler option. SIZE(MAX) indicates that the compiler obtains as much main storage in the region as it can. Since the compiler runs below the line, the storage obtained will be below the line. This can cause unpredictable problems as there will not be enough storage left for the system to use.

---

## Compiling for CICS

When coding a CICS transaction in PL/I, prior to compiling your transaction, you must invoke the CICS Command Language Translator. You can find information on the CICS Command Language Translator in the *CICS/ESA Application Programmer's Reference Manual*. After the CICS translator step ends, compile your PL/I program with the SYSTEM(CICS) option. NOEXECOPS is implied with this option. For a description of the SYSTEM compile-time option, see "SYSTEM" on page 24.

---

## Chapter 4. Compiling under VM

This chapter explains how to use the PLIOPT command to compile your program under VM. You must be linked to Language Environment before using PLIOPT, or your program will not compile. Language Environment must always be present when the PL/I compiler is active. The information in the chapter includes where the compiler stores its output, the types of files the compiler uses, and how to use the compile-time options. There is also information on special cases. The chapter describes how to include previously written PL/I statements with your program, compile your program to be run under MVS, and how to have your output placed in a TXTLIB. At the end of the chapter there are examples of PL/I batched compilation and information on compiler-detected errors.

To compile a program under VM, use the PLIOPT command followed by the name of the file that contains the source program. If the file type is not PLIOPT or PLI, you must specify the file type. If the file is not on the A disk, you must also specify the filemode naming the disk where the file is stored.

“PLIOPT Command Format” on page 76 shows the syntax for the PLIOPT command. If you want to specify any compile-time or PLIOPT options, these must follow the file name, file type, or file mode, whichever is the last you specified. You must put a left parenthesis before these options. Options are separated from each other by blanks, and you should use the abbreviated form of options.

During compilation, two new disk files are produced with the file types TEXT and LISTING and the same file name as the file specified in the PLIOPT command. The TEXT file contains the object code. The LISTING file contains the listings produced during compilation. Any error messages produced are transmitted to your terminal and contained in your listing.

If compilation reveals source program errors, you can alter the PLIOPT file that contains the source by use of the VM editor. You can then reissue the PLIOPT command. This results in the creation of new TEXT and LISTING files corresponding to the newly edited source programs. If previous versions were available they are overwritten. When you have a satisfactory compilation, you can run the program, which is now in the form of a TEXT file.

---

### Using the PLIOPT Command

Invoke the compiler by issuing the PLIOPT command. The compiler creates two output files. One file contains the object code, and the other file contains the listing. Refer to Table 3 on page 5 for a listing of compile-time options and their IBM-supplied defaults.

### Compiler Output and Its Destination

The compiler creates two new files and places them on VM disks by default. These files have the same file name as the file that contains the source type TEXT and the listing has the file type LISTING. Thus, if you compiled a PLIOPT file called ROBIN you would, by default, create two more files called ROBIN TEXT which contains the object code and ROBIN LISTING which contains the listing information. These files would be placed on your VM disks according to the rules

shown in Table 13. (The relationship between VM disks is explained in the *VM/ESA: CMS User's Guide*.)

It is possible to specify a name for the TEXT file other than that of the file compiled in the PLIOPT command by specifying a filename with the OBJECT option.

The creation of the LISTING file can be suppressed by use of the NOPRINT option of the PLIOPT command. (See "PLIOPT Command Options" on page 75.) The creation of the TEXT file can be suppressed by use of the NOOBJECT option of the PLIOPT command.

*Table 13. The disks on Which the Compiler Output Is Stored*

<b>If the disk that contains the PL/I source file is accessed...</b>	<b>then the disk that contains the output files (TEXT, LISTING) is:</b>
Read/Write...	the disk that holds the PL/I source.
as an extension of a Read/Write disk...	the Read/Write Disk.
as an extension of a Read-only Disk and the A-disk is accessed Read/Write...	the A-disk.
as an extension of a Read-only Disk and the A-disk is accessed Read Only...	ERROR DMSPLI006E — program terminates.

## Compile-Time Options

The PLIOPT command expects all options to be a maximum of eight characters long. You should always use the abbreviated form of the options. All options and suboptions must be separated by blanks. Parentheses need not be separated from options or suboptions even if the option has a total length of more than eight characters. Thus TERMINAL(XREF) is acceptable, although the total length is greater than eight characters.

Where options of the PLIOPT command contradict those of the %PROCESS statement, the options in the %PROCESS statement override those in the PLIOPT command. For options whose length is greater than eight characters, the abbreviation for that option must be used in the PLIOPT command.

## Files Used by the Compiler

During compilation the compiler uses a number of files. These files are allocated by the interface module that invokes the compiler. The files used are shown in Table 14. At the end of the compilation, the interface module will issue a FILEDEF \* CLEAR command to clear the definition of these files. As a result, all your file definitions without the PERM option active prior to the compilation will also be cleared.

*Table 14 (Page 1 of 2). Files That Can Be Used by the Compiler*

<b>FILE TYPE</b>	<b>FUNCTION</b>	<b>DEVICE TYPE</b>	<b>WHEN REQUIRED</b>
PLIOPT or PLI	Input	DASD, magnetic tape, card reader	Always
LISTING	Print	DASD, magnetic tape, printer	Optional
TEXT	Object module output	DASD, magnetic tape	When object module is to be created

Table 14 (Page 2 of 2). Files That Can Be Used by the Compiler

FILE TYPE	FUNCTION	DEVICE TYPE	WHEN REQUIRED
SYSPUNCH	System punch	DASD, magnetic tape, card punch	When MDECK and/or DECK is in effect
SYSUT1	Spill	DASD	When insufficient main storage is available
MACLIB	Preprocessor %INCLUDE	DASD	When %INCLUDE is used from VM disks
SYSLIB	Preprocessor %INCLUDE	DASD	When %INCLUDE is used from PL/I Library

## PLIOPT Command Options

The PLIOPT command compiles a PL/I program or a series of PL/I programs into machine language object code. If the file type is missing, the file type defaults to PLIOPT or PLI.

The following options are applicable only to the PLIOPT command and cannot appear on the %PROCESS statement in the PL/I source file.

- PRINT—The listing file is directed to the PRINTER and is not placed on a disk.
- DISK—The listing file is placed on a disk. To determine which disk, see Table 13 on page 74.
- TYPE—The listing file is displayed at your terminal and is not placed on a disk.
- NOPRINT—A listing file is not produced.
- OBJECT—An additional facility, OBJECT[(file name)], allows you to specify a different file name for your file.

In the OBJECT option specification, (file name) is the name that will be given to the text file. If it is omitted, the text file will be given the same name as the file specified in the PLIOPT command. The TEXT file will be placed on one of your disks in accordance with the rules shown in Table 13 on page 74.

### %INCLUDE Statement

If you want to use the %INCLUDE statement within your PL/I program, you must take the following steps:

- Create the file that you want to INCLUDE into your PL/I program. The file type must be COPY.
- Put the COPY file into an existing or new macro library (MACLIB).
- Use the %INCLUDE statement in your PL/I program.
- Issue a FILEDEF for the MACLIB that contains the COPY file you want included.
- If you have only %INCLUDE and no other preprocessor statements, compile your program using the compile-time option INC. If you have other preprocessor statements, use the compile time option MACRO.

The syntax of %INCLUDE is:

```
%INCLUDE DDNAME(member name);
```

The following example demonstrates the use of the %INCLUDE statement.

## Example of Using %INCLUDE

The COPY file called PLIXOPT COPY is created:

```
DCL PLIXOPT CHAR(255) VAR STATIC EXTERNAL
  INIT ('STACK(4K),HEAP(4K),RPTSTG(ON)');
```

The COPY file PLIXOPT is added to the MACLIB called MYLIB:

```
MACLIB ADD MYLIB PLIXOPT
```

If a MACLIB does not exist, use the command MACLIB GEN instead of MACLIB ADD. This will generate a MACLIB called MYLIB.

In the PL/I source file, the following %INCLUDE statement is included:

```
%INCLUDE PLICOPY(PLIXOPT);
```

A FILEDEF is issued for the ddname specified in the %INCLUDE statement to tell PL/I where to obtain the member PLIXOPT within a library:

```
FILEDEF PLICOPY DISK MYLIB MACLIB
```

The PL/I program is compiled. The program has no other preprocessor statements, so the INC option is used:

```
PLIOPT EXAMPLE ( INC
```

For complete information on the VM Commands which are used above, see the *VM/ESA: CMS Command Reference*.

## PLIOPT Command Format

The format of the PLIOPT command is:

```
PLIOPT filename[filetype[filemode]] [(options-list [])]
```

where filename[filetype[filemode]] is the identification of the file that contains the PL/I source program. If filetype is omitted, a search will be made first for PLIOPT files of the specified filename and then for PLI files of the specified filename. If filemode is omitted, A will be assumed.

If the options list is (option1 option2 option3... then the options must be separated from each other by at least one blank. The right hand parenthesis is optional. If contradicting options are specified, the rightmost option applies. See Table 3 on page 5 for information on options and their correct syntax.

### Examples:

To compile a PLIOPT or PLI file called RABBIT on the A-disk with the OPTIONS and SOURCE options:

```
PLIOPT RABBIT (OPTIONS SOURCE
```

To compile a file with the name RABBIT and the type FORMAT on the B-disk with the options PRINT, XREF, and ATTRIBUTES:

```
PLIOPT RABBIT FORMAT B (PRI X A
```

Note that the abbreviations are used for these options.

## Special Action Will Be Required:

1. If your source uses the %INCLUDE statement to incorporate secondary input text.
2. If you intend to run your program under MVS.
3. If you want to place the compiled program into a TXTLIB. You might want to do this if you want to use separately compiled subroutines.

The following paragraphs describe the actions required in each of these circumstances.

**Using %INCLUDE under VM:** If your program uses %INCLUDE statements to include previously written PL/I statements or procedures, the libraries on which they are held must be made available to VM before issuing the PLIOPT command. To do this you must insert the statements into a VM MACLIB using the MACLIB command. You then issue a GLOBAL command taking the form "GLOBAL MACLIB filename."

For example, if your secondary input text was held in MACLIB called "MYLIB" you would enter:

```
GLOBAL MACLIB MYLIB
```

before issuing the PLIOPT command. The PLIOPT command must specify either the INCLUDE or the MACRO option.

If your %INCLUDE statement takes the form %INCLUDE MYLIB (CUCKOO), as opposed to %INCLUDE CUCKOO, you will also need to specify a FILEDEF command for MYLIB. This should take the form:

```
FILEDEF MYLIB DISK MYLIB MACLIB
```

If in the MACLIB the LRECL is not 80 and the BLOCKSIZE not 400, format information must be included in the FILEDEF command.

**Compiling a Program to Run under MVS:** If you intend to run your program under MVS, you should specify the SYSTEM(MVS) option:

```
PLIOPT RABBIT (SYSTEM(MVS))
```

An attempt to run a program compiled without the SYSTEM(MVS) option under MVS results in an OS linkage editor error of severity level 8.

**Compiling a Program to be Placed in a TXTLIB:** If you intend to include the compiled TEXT file as a member of a TXTLIB it is necessary to use the NAME option when you specify the PLIOPT command. This is because members of a TXTLIB file are given the name of their primary entry point if they have no external name. The primary entry point of every TEXT file produced by the compiler is the same, consequently only one compiled program can be included in a TXTLIB if the NAME option is not used. (The NAME option gives the TEXT file an external name.)

Commands required to create a TEXT file suitable for including in a TXTLIB are shown below. This code gives the file the external name used in the PLIOPT command. However, any other name can be used provided that it does not exceed six characters.

**Note:** If the name exceeds six characters, the NAME option is ignored.

The commands below compile a PLIOPT file RABBIT with the external name RABBIT and add it to an existing text library called BIOLIB.

```
PLIOPT RABBIT (NAME('RABBIT'  
[compiler messages etc.]  
TXTLIB ADD BIOLIB RABBIT
```

If the BIOLIB TXTLIB does not exist yet, use the command TXTLIB GEN instead of TXTLIB ADD.

## PL/I Batched Compilation

An example of VM batched compilation is shown in Figure 15.

---

```
PLIOPT FIRST  
  
where FIRST and SECND are a single file that looks like:  
  first: proc;  
  .  
  .  
  .  
  end;  
%process;  
  secnd:proc;  
  .  
  .  
  .  
  end;
```

---

Figure 15. Example of Batched Compilation under VM

---

## Correcting Compiler-Detected Errors

At compile time, both the preprocessor and the compiler can produce diagnostic messages and listings according to the compile-time options selected for a particular compilation. The listings and the associated compile-time options are discussed in Chapter 1, “Using Compile-Time Options and Facilities” on page 5. The diagnostic messages produced by the compiler are identified by a number with an “IEL” prefix. These diagnostic messages are available in both a long form and a short form. The short messages are obtained by specifying the SMESSAGE compiler option. Each message is listed in *PL/I for MVS & VM Compile-Time Messages and Codes*. This publication includes explanatory notes, examples, and any action to be taken.

Always check the compilation listing for occurrences of these messages to determine whether the syntax of the program is correct. Messages of greater severity than warning (that is, error, severe error, and unrecoverable error) should be acted upon if the message does not indicate that the compiler has been able to “fix” the error correctly. You should be aware that the compiler, in making an assumption as to the intended meaning of any erroneous statement in the source program, can introduce another, perhaps more severe, error which in turn can produce yet another error, and so on. When this occurs, the compiler produces a number of diagnostic messages which are all caused either directly or indirectly by the one error.

Other useful diagnostic aids produced by the compiler are the attribute table and cross-reference table. The attribute table, specified by the ATTRIBUTES option, is useful for checking that program identifiers, especially those whose attributes are contextually and implicitly declared, have the correct attributes. The cross-reference table is requested by the XREF option, and indicates, for each program variable, the number of each statement that refers to the variable.

To prevent unnecessary waste of time and resources during the early stages of developing programs, use the NOOPTIMIZE, NOSYNTAX, and NOCOMPILE options. The NOOPTIMIZE option suppresses optimization unconditionally, and the remaining options suppress compilation, link-editing, and execution if the appropriate error conditions are detected.

---

## Chapter 5. Link-Editing and Running

After compilation, your program consists of one or more object modules that contain unresolved references to each other, as well as references to the Language Environment for MVS & VM run-time library. These references are resolved during link-editing or during execution (dynamically).

So after you compile your PL/I program, the next step is to link and run your program with test data to verify that it produces the results you expect.

Language Environment for MVS & VM provides the run-time environment and services you need to execute your program. For instructions on linking and running PL/I and all other Language Environment for MVS & VM-conforming language programs, refer to the *Language Environment for MVS & VM Programming Guide*. For information about migrating your existing PL/I programs to Language Environment for MVS & VM, see the *PL/I for MVS & VM Compiler and Run-Time Migration Guide*.

This chapter contains the following sections:

- Selecting math results at link-edit time
- VM run-time considerations
- MVS run-time considerations
- SYSPRINT Considerations

### Selecting Math Results at Link-Edit Time

You can select math results that are compatible with Language Environment for MVS & VM or with OS PL/I. When you link your load module, you select the math results by linking in the stubs for the Language Environment for MVS & VM math routines or the OS PL/I math routines. You select the results on a load module basis; a load module that uses the Language Environment for MVS & VM results can fetch a load module that uses the OS PL/I results.

Because the Language Environment for MVS & VM routines are defaults, if you relink an OS PL/I application, you receive the Language Environment for MVS & VM results. To maintain the OS PL/I results, you need to ensure that the stubs for the PL/I math routines are linked into the application. You can do so by overriding the linkedit library SYSLIB data set name with the name of the PL/I math link-edit library data set, SIBMMATH.

Use the following JCL or equivalent:

```
//SYSLIB DD DSN=CEE.V1R2M0.SIBMMATH,DISP=SHR
//      DD DSN=CEE.V1R2M0.SCEELKED,DISP=SHR
```

---

### VM Run-Time Considerations

Various special topics are covered in this section, including PL/I restrictions under VM.

## Separately Compiled PL/I MAIN Programs

You can load separately compiled procedures with the MAIN option into one executable program in PL/I. The PL/I procedure that you want to receive control first must be specified first on the LOAD command. For example, if you have two MAIN PL/I procedures CALLING and CALLED (see Figure 16 and Figure 17) and you want CALLING to receive control first, you issue these VM commands:

```
global txtlib plilib sceelked cmslib /* make the libraries available */
Ready;
pliopt calling (system(cms)          /* compile the one of the procs */
Ready;
pliopt called (system(cms)          /* compile the other one      */
Ready;
global loadlib sceerun              /* make the libraries available */
load calling called ( nodup         /* CALLING will receive control */
Ready;                               /* ...first. NODUP suppresses */
                                       /* ...duplicate identifier msgs */
start                               /* invoke the program          */
Ready;
```

---

```
%PROCESS F(I) AG A(F) ESD MAP OP STG NEST X(F) SOURCE LIST ;
Calling: Proc Options(Main);
Dcl Sysprint File Output;
Dcl Called External Entry;

Put Skip List ('CALLING - started');
Call Called;
Put Skip List ('CALLING - Ended');

END Calling;
```

Figure 16. PL/I Main Calling Another PL/I Main

---

```
%PROCESS F(I) AG A(F) ESD MAP OP STG TEST X(F) SOURCE LIST ;
Called: Proc Options(Main);
Dcl Sysprint File ;

Put Skip List ('CALLED - started');
Put Skip List ('CALLED - ended');

END Called;
```

Figure 17. PL/I Main Called by Another PL/I Main

## Using Data Sets and Files

VM files and other OS data sets can be written and read by programs run under VM, with varying restrictions.

VM files are completely accessible for read, write, and update to programs running under VM. You can make these files available to a number of virtual machines, but they are not accessible from outside the VM system except by copying and recreation.

Only sequential OS data sets are available, on a read-only basis, to VM programs.

Within a program, a file is identified by the declared name or the name given in the title option. Outside the program, the FILEDEF command, or the DLBL command for VSAM, binds a file name to a particular data set.

VSAM data sets are different from other types of files because their management is handled by a set of programs known as Access Method Services. The services are available to the VM user by the AMSERV command. This command uses a previously created file containing Access Method Services statements to specify the required services.

VM uses the DOS data management routines which must be installed during VM program installation. Your program is not affected by the use of DOS routines, but certain OS Access Method Services functions are not available for data set handling. Full details of this and other aspects of VM VSAM are given in the *VM/ESA CMS User's Guide*.

To test programs that create or modify OS data sets, you can write "OS-Simulated data sets." These are VM files that are maintained on VM disks in OS format, rather than in VM format. You can perform any VM file operation on these files. However, since they are in the OS-Simulated format, files with variable-blocked records can contain block and record descriptor words, so that the access methods can manipulate the files properly. If you specify the filemode number as 4, VM creates a file that is in OS-Simulated data set format.

The following three examples show the PL/I statements and the CMS commands necessary to access VM files, VSAM data sets, and non-VSAM OS data sets, respectively.

### **Using VM Files — Example**

To access a VM file, issue a FILEDEF command associating a PL/I file name with particular VM file(s).

In the example that follows, the PL/I program reads the file known in the program as "OLDFILE". This refers to the VM file "INPUT DATA B". The program creates the file known in the program as "NEWFILE", which corresponds to the VM file "OUTPUT DATA A". A third file, PL/I file "HISTORY", is assigned to the virtual printer.

#### ***PL/I Program Statements***

```
DCL OLDFILE FILE RECORD INPUT ENV (F RECSIZE(40)),  
      NEWFILE FILE RECORD OUTPUT ENV (F RECSIZE(40)),  
      HISTORY FILE STREAM PRINT;
```

## **VM Commands**

<code>filedef oldfile disk input data b</code>	Associates OLDFILE with the file INPUT DATA B.
<code>filedef newfile disk output data a</code>	Associates NEWFILE with the file OUTPUT DATA A.
<code>filedef history printer</code>	Associates the file HISTORY with the virtual printer.

The full syntax of the FILEDEF and other commands is given in *VM/ESA CMS Command Reference*.

## **Using VSAM Data Sets — Example**

VSAM data sets differ from other data sets because they are always accessed through a catalog and because they have their routine management performed by Access Method Services. Use the AMSERV command to invoke Access Method Services functions and the DLBL command to associate an actual VSAM data set with the file identifier in a PL/I program.

To use the AMSERV command, a file of the filetype AMSERV must be created that contains the necessary Access Method Services commands. An AMSERV command, specifying the name of this file, is then issued and the requested Access Method Services are performed. Such services must always be used for cataloging and formatting purposes before creating a VSAM data set. They are also used for deleting, renaming, making portable copies, and other routine tasks.

For VSAM data sets, catalog entries are created by the DEFINE statement of Access Method Services. They contain information such as the space used or reserved for the data set, the record size, and the position of a key within the record. The catalog entry also contains the address of the data set.

To use a VSAM data set, you must identify the catalog to be searched and associate the PL/I file with the VSAM data set. The DLBL command is used for both these purposes. Where the data set is being newly created, you must specify the AMSERV command to catalog and define the data set before the PL/I program is executed. Details of how to use VSAM under VM are given in the *VM/ESA CMS User's Guide*.

The relevant PL/I statements and VM commands to access an existing VSAM data set and to create a new VSAM data set are shown in the example that follows.

The PL/I program reads the file OLDRAB from the VSAM data set called RABBIT1 on the VM B-disk. It writes the file NEWRAB onto the data set RABBIT2, also on the VM B-disk. RABBIT2 is defined using an AMSERV command. In the example, this master catalog is already assigned and the VSAM space is also already assigned.

### **PL/I File Declaration**

```
DCL OLDRAB FILE RECORD SEQUENTIAL KEYED INPUT ENV(VSAM);  
DCL NEWRAB FILE RECORD SEQUENTIAL KEYED OUTPUT ENV(VSAM);
```

**VM Commands:** A file with the filetype of AMSERV must be created with the appropriate Access Method Services commands, and is named 'AMSIN AMSERV'. For this example, the file must contain the following information:

```

DEFINE CLUSTER(NAME(RABBIT2.C) VOL(VOLSER)) -
  DATA (CYL(4,1) KEYS(5,5) RECSZ(23,23) -
  FREESPACE(20,30)) -
  INDEX(CYL(1,1))

```

The VM commands that you need to issue are:

dlbl ijsyscat b dsn mastca (perm	Issue a DLBL for the master catalog. Note that this need only be done once for terminal session if PERM is specified.
amserv amsin	Execute statements in the AMSERV file to catalog and format data set.
dlbl oldrab b dsn rabbit1 (vsam)	Issue DLBL commands to associate PL/I files with the VSAM data sets.
dlbl newrab b dsn rabbit2 (vsam)	

**Notes:**

1. The closing parenthesis is optional in VM commands but *required* in Access Method Services commands.
2. PL/I MVS & VM programs with files declared with ENV(INDEXED) can, in certain instances, operate correctly if the data set being accessed is a VSAM data set.

**Using OS Data Sets — Example**

Before you can access an OS data set that resides on an OS formatted disk, it must be made available to your virtual machine. Using the ACCESS command, you can access the OS formatted disk as one of your VM minidisks. Once this has been done, you can use a FILEDEF command to access the disk in the usual manner.

In the example that follows, the PL/I file OLDRAB is used to access the OS data set RABBIT.OS.DATA. The disk containing the data set has been mounted and is known to the user as virtual disk number 196.

**PL/I Statement**

```
DCL OLDRAB FILE RECORD ENV (F RECSIZE(40));
```

**VM Commands**

access 196 g DMSACP723I G (196) R/O	Connect disk containing data set to your virtual machine.
filedef oldrab g dsn rabbit os data	Associate PL/I file OLDRAB with OS data set RABBIT.OS.DATA.

**Using Tapes with Standard Labels:** VM assumes that tapes do not have standard labels. If you want to process a standard label tape, you can use the VM commands LABELDEF, FILEDEF, and/or TAPE. More information can be found in the *VM/ESA CMS Command Reference*.

## Restrictions Using PL/I under VM

PL/I features that are not available under VM are:

- ASCII data sets
- BACKWARDS attribute with magnetic tapes
- INDEXED Files (except for use with VSAM)
- PL/I checkpoint restart facilities (PLICKPT)
- Tasking
- Regional(2) and Regional(3) files
- Teleprocessing\* files (TCAM)
- VS or VBS record formats.

PL/I features that have restricted use under VM are:

Regional(1) files      Regional(1) files can be used with the following restrictions:

- More than one regional file with keys cannot be open at the same time.
- KEY(TRACKID/REGION NUMBER) must not be incremented unless 255 records are written on the first logical track, and 256 records on each subsequent logical track.
- Files must not be written with a dependency on the physical track length of a direct access device.
- When a file is created, the XTENT option of the FILEDEF command must be specified, and it must be equal to the number of records in the file to be created.

READ                      This can only be used if the NCP parameter is included in the ENVIRONMENT option of the PL/I file.

Blanks                    Blanks cannot be passed in the parameter string to the main procedure using SYSTEM(CMSTPL). The blanks are removed from the string and the items separated by them are concatenated. Use of SYSTEM(CMS) does not have this restriction.

TIME                      The TIME built-in function returns values calculated to the nearest second.

VSAM                      VSAM data sets can be used only if DOS/VS VSAM was incorporated into VM during PL/I VM installation. DOS VSAM is used and any features not available to DOS VSAM cannot be used. CMS/DOS must also be generated into VM. For details of how to do this, see *VM/ESA Installation*.

Environment options: SIS cannot be used, SKIP cannot be used on ESDS.

### Using Record I/O at the Terminal

There is no provision for input prompting or synchronization of output for RECORD files assigned to the terminal. Terminal interaction logic is generally easier to write using stream I/O, but when you use record I/O at the terminal, keep the following points in mind:

**Output:** Output files should be declared with BUFFERS(1) if you must synchronize input with output.

Use V-format records; otherwise trailing blanks are transmitted.

**Input:** Use V-format records, as doing otherwise raises the RECORD condition unless the record is filled out with trailing blanks. Note that when V-format records are used and the data is read into a fixed length string, the string is not padded with blanks. By default, RECORD files assigned to the terminal are given F-format records with the record length the same as the linesize for the terminal.

## PL/I Conventions under VM

Two types of conventions apply to PL/I when used under VM. The first type is adopted to make input/output simpler and more efficient at the terminal. The second type results from the terminal being considered as the console of a virtual machine. These affect the DISPLAY statement and the REPLY option.

**Stream I/O Conventions at the Terminal:** To simplify input/output at the terminal, various conventions have been adopted for stream files that are assigned to the terminal. Three areas are affected:

1. Formatting of PRINT files
2. The automatic prompting feature
3. Spacing and punctuation rules for input.

### Formatting Conventions for PRINT Files

When a PRINT file is assigned to the terminal, it is assumed that it will be read as it is being printed. Spacing is therefore reduced to a minimum to reduce printing time. The following rules apply to the PAGE, SKIP, and ENDPAGE keywords:

- PAGE options or format items result in three lines being skipped.
- SKIP options or format items large than SKIP (2) result in three lines being skipped. SKIP (2) or less is treated in the usual manner.
- The ENDPAGE condition is never raised.

### Changing the Format on PRINT Files

If you want normal spacing to apply to output from a PRINT file at the terminal, you must supply your own tab table for PL/I. This is done by declaring an external structure called PLITABS in the program and initializing the element PAGELENGTH to the number of lines that can fit on your page. This value differs from PAGESIZE, which defines the number of lines you want to be printed on the page before ENDPAGE is raised. (See Figure 18 and Figure 19 on page 90 in "MVS Run-Time Considerations.")

### Automatic Prompting

When the program requires input from a file that is associated with a terminal, it issues a prompt. This takes the form of printing a colon on the next line and then skipping to column 1 on the line following the colon. This gives you a full line to enter your input, as follows:

```
:  
(space for entry of your data)
```

This type of prompt is referred to as a primary prompt.

**Overriding Automatic Prompting:** It is possible to override the primary prompt by making a colon the last item in the request for the data. The secondary prompt cannot be overridden. For example, the two PL/I statements:

```
PUT SKIP EDIT ('ENTER TIME OF PERIHELION') (A);
GET EDIT (PERITIME) (A(10));
```

result in the terminal printing:

```
ENTER TIME OF PERIHELION
:          (automatic prompt)
(space for entry of data)
```

However, if the first statement has a colon at the end of the output, as follows:

```
PUT EDIT ('ENTER TIME OF PERIHELION:') (A);
```

the sequence is:

```
ENTER TIME OF PERIHELION: (space for entry of data)
```

**Note:** The override remains in force for only one prompt. You will be automatically prompted for the next item unless the automatic prompt is again overridden.

## Punctuating Long Input Lines

**Line Continuation Character:** To transmit data that requires 2 or more lines of space at the terminal as one data-item, type an SBCS hyphen as the last character in each line except the last line. For example, to transmit the sentence “this data must be transmitted as one unit.” you enter:

```
:
' this data must be transmitted -
:
as one unit.'
```

Transmission does not occur until you press ENTER after “unit.” The hyphen is removed. The item transmitted is called a “logical line.”

**Note:** To transmit a line whose last data character is a hyphen or a PL/I minus sign, enter two hyphens at the end of the line, followed by a null line as the next line.

For example:

```
xyz--  
(press ENTER only, on this line)
```

### **Punctuating GET LIST and GET DATA Statements**

For GET LIST and GET DATA statements, a comma is added to the end of each logical line transmitted from the terminal, if the programmer omitted it. Thus there is no need to enter blanks or commas to delimit items if they are entered on separate logical lines. For the PL/I statement GET LIST(A,B,C); you can enter at the terminal:

```
:  
 1  
+:  
 2  
+:  
 3
```

This rule also applies when entering character-string data. A character string must therefore transmit as one logical line. Otherwise, commas are placed at the break points. For example, if you enter:

```
:  
'COMMAS SHOULD NOT BREAK  
+:  
  UP A CLAUSE.'
```

the resulting string is "COMMAS SHOULD NOT BREAK, UP A CLAUSE." The comma is not added if a hyphen was used as a line continuation character.

**Automatic Padding for GET EDIT:** For a GET EDIT statement, there is no need to enter blanks at the end of the line. The data will be padded to the specified length. Thus, for the PL/I statement:

```
GET EDIT (NAME) (A(15));
```

you can enter the 5 characters SMITH. The data will be padded with ten blanks so that the program receives the fifteen characters:

```
'SMITH          '
```

**Note:** A single data item must transmit as a logical line. Otherwise, the first line transmitted will be padded with the necessary blanks and taken as the complete data item.

**Use of SKIP for Terminal Input:** All uses of SKIP for input are interpreted as SKIP(1) when the file is allocated to the terminal. SKIP(1) is treated as an instruction to ignore all unused data on the currently available logical line.

### **ENDFILE**

The end-of-file can be entered at the terminal by keying in a logical line that consists of the two characters "/\*". Any further attempts to use the file without closing it result in the ENDFILE condition being raised.

## DISPLAY and REPLY under VM

Because your terminal is the console of the virtual machine, you can use the DISPLAY statement and the REPLY option to create conversational programs. The DISPLAY statement transmits the message to your terminal, and the REPLY option allows you to respond. For example, the PL/I statement:

```
DISPLAY ('ENTER NAME') REPLY (NAME);
```

results in the message “ENTER NAME” being printed at your terminal. The program then waits for your response and places your data in the variable NAME after you press ENTER. The terminal display looks like:

```
ENTER NAME  
Esther Summers
```

The reply can contain DBCS characters but they must be processable as a mixed string.

**Note:** File I/O can be buffered if the file is directed to the terminal. If you are using I/O directed to the terminal as well as the DISPLAY statement, the order of the lines written cannot be the same as the program intended.

---

## MVS Run-Time Considerations

To simplify input/output at the terminal, various conventions have been adopted for stream files that are assigned to the terminal. Three areas are affected:

1. Formatting of PRINT files
2. The automatic prompting feature
3. Spacing and punctuation rules for input.

**Note:** No prompting or other facilities are provided for record I/O at the terminal, so you are strongly advised to use stream I/O for any transmission to or from a terminal.

## Formatting Conventions for PRINT Files

When a PRINT file is assigned to the terminal, it is assumed that it will be read as it is being printed. Spacing is therefore reduced to a minimum to reduce printing time. The following rules apply to the PAGE, SKIP, and ENDPAGE keywords:

- PAGE options or format items result in three lines being skipped.
- SKIP options or format items larger than SKIP (2) result in three lines being skipped. SKIP (2) or less is treated in the usual manner.
- The ENDPAGE condition is never raised.

## Changing the Format on PRINT Files

If you want normal spacing to apply to output from a PRINT file at the terminal, you must supply your own tab table for PL/I. This is done by declaring an external structure called PLITABS in the program and initializing the element PAGEDLENGTH to the number of lines that can fit on your page. This value differs from PAGESIZE, which defines the number of lines you want to print on the page before ENDPAGE is raised (see Figure 19 on page 90). If you require a PAGEDLENGTH of 64 lines, declare PLITABS as shown in Figure 18 on page 90. For information on overriding the tab table, see “Overriding the Tab Control Table” on page 140.

---

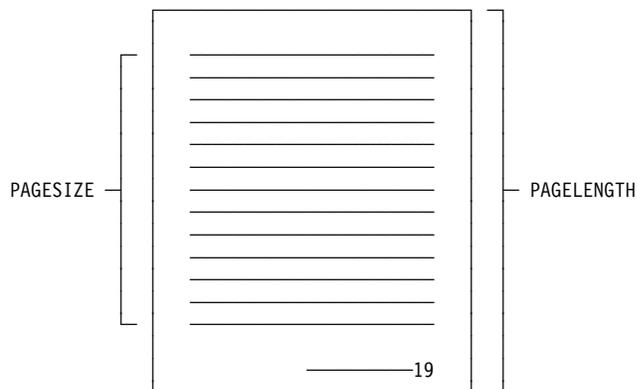
```

DCL 1 PLITABS STATIC EXTERNAL,
  ( 2  OFFSET INIT (14),
    2  PAGESIZE INIT (60),
    2  LINESIZE INIT (120),
    2  PAGELength INIT (64),
    2  FILL1 INIT (0),
    2  FILL2 INIT (0),
    2  FILL3 INIT (0),
    2  NUMBER_OF_TABS INIT (5),
    2  TAB1 INIT (25),
    2  TAB2 INIT (49),
    2  TAB3 INIT (73),
    2  TAB4 INIT (97),
    2  TAB5 INIT (121)) FIXED BIN (15,0);

```

---

Figure 18. Declaration of PLITABS. This declaration gives the standard page size, line size and tabulating positions



PAGELength: the number of lines that can be printed on a page

PAGESIZE: the number of lines that will be printed on a page before the ENDPAGE condition is raised

---

Figure 19. PAGELength and PAGESIZE. PAGELength defines the size of your paper, PAGESIZE the number of lines in the main printing area.

## Automatic Prompting

When the program requires input from a file that is associated with a terminal, it issues a prompt. This takes the form of printing a colon on the next line and then skipping to column 1 on the line following the colon. This gives you a full line to enter your input, as follows:

```

:
(space for entry of your data)

```

This type of prompt is referred to as a primary prompt.

**Overriding Automatic Prompting:** You can override the primary prompt by making a colon the last item in the request for the data. You cannot override the secondary prompt. For example, the two PL/I statements:

```

PUT SKIP EDIT ('ENTER TIME OF PERIHELION') (A);
GET EDIT (PERITIME) (A(10));

```

result in the terminal displaying:

```
ENTER TIME OF PERIHELION
:      (automatic prompt)
(space for entry of data)
```

However, if the first statement has a colon at the end of the output, as follows:

```
PUT EDIT ('ENTER TIME OF PERIHELION:') (A);
```

the sequence is:

```
ENTER TIME OF PERIHELION: (space for entry of data)
```

**Note:** The override remains in force for only one prompt. You will be automatically prompted for the next item unless the automatic prompt is again overridden.

## Punctuating Long Input Lines

**Line Continuation Character:** To transmit data that requires 2 or more lines of space at the terminal as one data-item, type an SBCS hyphen as the last character in each line except the last line. For example, to transmit the sentence “this data must be transmitted as one unit.” you enter:

```
: 'this data must be transmitted -
+: as one unit.'
```

Transmission does not occur until you press ENTER after “unit.’”. The hyphen is removed. The item transmitted is called a “logical line.”

**Note:** To transmit a line whose last data character is a hyphen or a PL/I minus sign, enter two hyphens at the end of the line, followed by a null line as the next line. For example:

```
xyz--
      (press ENTER only, on this line)
```

## Punctuating GET LIST and GET DATA Statements

For GET LIST and GET DATA statements, a comma is added to the end of each logical line transmitted from the terminal, if the programmer omits it. Thus there is no need to enter blanks or commas to delimit items if they are entered on separate logical lines. For the PL/I statement GET LIST(A,B,C); you can enter at the terminal:

```
:1
+:2
+:3
```

This rule also applies when entering character-string data. Therefore, a character string must transmit as one logical line. Otherwise, commas are placed at the break points. For example, if you enter:

```
: 'COMMAS SHOULD NOT BREAK
+: UP A CLAUSE.'
```

the resulting string is: “COMMAS SHOULD NOT BREAK, UP A CLAUSE.” The comma is not added if a hyphen was used as a line continuation character.

**Automatic Padding for GET EDIT:** For a GET EDIT statement, there is no need to enter blanks at the end of the line. The data will be padded to the specified length. Thus, for the PL/I statement:

```
GET EDIT (NAME) (A(15));
```

you can enter the 5 characters SMITH. The data will be padded with ten blanks so that the program receives the fifteen characters:

```
'SMITH          '
```

**Note:** A single data item must transmit as a logical line. Otherwise, the first line transmitted will be padded with the necessary blanks and taken as the complete data item.

**Use of SKIP for Terminal Input:** All uses of SKIP for input are interpreted as SKIP(1) when the file is allocated to the terminal. SKIP(1) is treated as an instruction to ignore all unused data on the currently available logical line.

## ENDFILE

The end-of-file can be entered at the terminal by keying in a logical line that consists of the two characters “/”. Any further attempts to use the file without closing it result in the ENDFILE condition being raised.

---

## SYSPRINT Considerations

The PL/I standard SYSPRINT file is shared by multiple enclaves within an application. You can issue I/O requests, for example STREAM PUT, from the same or different enclaves. These requests are handled using the standard PL/I SYSPRINT file as a file which is common to the entire application. The SYSPRINT file is implicitly closed only when the application terminates, not at the termination of the enclave.

The standard PL/I SYSPRINT file contains user-initiated output only, such as STREAM PUTs. Run-time library messages and other similar diagnostic output are directed to the Language Environment MSGFILE. See the *Language Environment for MVS & VM Programming Guide* for details on redirecting SYSPRINT file output to the Language Environment MSGFILE.

To be shared by multiple enclaves within an application, the PL/I SYSPRINT file must be declared as an EXTERNAL FILE constant with a file name of SYSPRINT and also have the attributes STREAM and OUTPUT as well as the (implied) attribute of PRINT, when OPENed. This is the standard SYSPRINT file as defaulted by the compiler.

There exists only one standard PL/I SYSPRINT FILE within an application and this file is shared by all enclaves within the application. For example, the SYSPRINT file can be shared by multiple nested enclaves within an application or by a series of enclaves that are created and terminated within an application by the Language Environment preinitialization function. To be shared by an enclave within an application, the PL/I SYSPRINT file must be declared in that enclave. The standard SYSPRINT file cannot be shared by passing it as a file argument between enclaves. The declared attributes of the standard SYSPRINT file should be the same throughout the application, as with any EXTERNALLY declared constant. PL/I does not enforce this rule.

Having a common SYSPRINT file within an application can be an advantage to applications that utilize enclaves that are closely tied together. However, since all enclaves in an application write to the same shared data set, this might require some coordination among the enclaves.

The SYSPRINT file is opened (implicitly or explicitly) when first referenced within an enclave of the application. When the SYSPRINT file is CLOSED, the file resources are released (as though the file had never been opened) and all enclaves are updated to reflect the closed status.

If SYSPRINT is utilized in a multiple enclave application, the LINENO built-in function only returns the current line number until after the first PUT or OPEN in an enclave has been issued. This is required in order to maintain full compatibility with old programs.

The COUNT built-in function is maintained at an enclave level. It always returns a value of zero until the first PUT in the enclave is issued. If a nested child enclave is invoked from a parent enclave, the value of the COUNT built-in function is undefined when the parent enclave regains control from the child enclave.

When opened, the TITLE option can be used to associate the standard SYSPRINT file with different operating system data sets. This association is retained across enclaves for the duration of the open.

PL/I condition handling associated with the standard PL/I SYSPRINT file retains its current semantics and scope. For example, an ENDPAGE condition raised within a child enclave will only invoke an established on-unit within that child enclave. It does not cause invocation of an on-unit within the parent enclave.

The tabs for the standard PL/I SYSPRINT file can vary when PUTs are done from different enclaves, if the enclaves contain a user PLITABS table.

OS PL/I I/O FETCH/RELEASE restrictions continue to apply to the SYSPRINT file. If SYSPRINT is declared as a file constant in a load module, the declared SYSPRINT file information is statically and locally bound to that load module and the following rules apply:

- If the load module has been released from storage, explicit use of this file constant by the user program can cause unpredictable results.
- If file comparison or I/O on-units are involved, the use of these language features is scoped to the load module.

For example, if SYSPRINT is declared in load module A and also in load module B, a file comparison of the two SYSPRINTs will not compare equal. Similarly, if an ENDPAGE on-unit for SYSPRINT is established in load module A and a PUT is done in load module B, the ENDPAGE on-unit might not gain control if the PUT overflows a page.

The scoping rules for file comparison and I/O units can be avoided if you declare SYSPRINT as a file constant in a particular load module and use a file variable parameter to pass that SYSPRINT declaration to other load modules for file comparison or PUTs. In this case, the load module boundary scoping rules do not apply.

When the PL/I SYSPRINT file is used with the PL/I multitasking facility, the task-level file-sharing rules apply. This maintains full compatibility for old PL/I multitasking programs.

If the PL/I SYSPRINT file is utilized as a RECORD file or as a STREAM INPUT file, PL/I supports it at an individual enclave or task level, but not as a sharable file among enclaves. If the PL/I SYSPRINT file is open at the same time with different file attributes (e.g. RECORD and STREAM) in different enclaves of the same application, results are unpredictable.

---

## Part 3. Using I/O facilities

<b>Chapter 6. Using Data Sets and Files</b> .....	99
Associating Data Sets with Files .....	99
Associating Several Files with One Data Set .....	101
Associating Several Data Sets with One File .....	102
Concatenating Several Data Sets .....	102
Establishing Data Set Characteristics .....	102
Blocks and Records .....	103
Record Formats .....	103
Fixed-Length Records .....	104
Variable-Length Records .....	104
Undefined-Length Records .....	106
Data Set Organization .....	106
Labels .....	107
Data Definition (DD) Statement .....	107
Use of the Conditional Subparameters .....	108
Data Set Characteristics .....	108
Associating PL/I Files with Data Sets .....	109
Specifying Characteristics in the ENVIRONMENT Attribute .....	110
Data Set Types Used by PL/I Record I/O .....	121
<b>Chapter 7. Using Libraries</b> .....	123
Types of libraries .....	123
How to Use a Library .....	123
Creating a Library .....	124
SPACE Parameter .....	124
Creating and Updating a Library Member .....	125
Examples .....	125
Extracting Information from a Library Directory .....	128
<b>Chapter 8. Defining and Using Consecutive Data Sets</b> .....	129
Using Stream-Oriented Data Transmission .....	129
Defining Files Using Stream I/O .....	130
Specifying ENVIRONMENT Options .....	130
CONSECUTIVE .....	130
Record format options .....	130
RECSIZE .....	131
Defaults for Record Format, BLKSIZE, and RECSIZE .....	131
GRAPHIC Option .....	132
Creating a Data Set with Stream I/O .....	132
Essential Information .....	132
Examples .....	133
Accessing a Data Set with Stream I/O .....	136
Essential Information .....	136
Record Format .....	137
Example .....	137
Using PRINT Files with Stream I/O .....	138
Controlling Printed Line Length .....	139
Overriding the Tab Control Table .....	140
Using SYSIN and SYSPRINT Files .....	142
Controlling Input from the Terminal .....	143

Using Files Conversationally	143
Format of Data	143
Stream and Record Files	144
Capital and Lowercase Letters	145
End-of-File	145
COPY Option of GET Statement	145
Controlling Output to the Terminal	145
Format of PRINT Files	145
Stream and Record Files	146
Capital and Lowercase Characters	146
Output from the PUT EDIT Command	146
Example of an Interactive Program	146
Using Record-Oriented Data Transmission	149
Using Magnetic Tape without Standard Labels	150
Specifying Record Format	150
Defining Files Using Record I/O	150
Specifying ENVIRONMENT Options	150
CONSECUTIVE	151
TOTAL	151
CTLASA CTL360	152
LEAVE REREAD	153
ASCII	154
BUFOFF	155
D-Format and DB-Format Records	155
Creating a Data Set with Record I/O	156
Essential Information	156
Accessing and Updating a Data Set with Record I/O	157
Essential Information	159
Example of Consecutive Data Sets	159
<b>Chapter 9. Defining and Using Indexed Data Sets</b>	<b>163</b>
Indexed Organization	163
Using keys	163
Using Indexes	166
Dummy Records	167
Defining Files for an Indexed Data Set	169
Specifying ENVIRONMENT Options	169
ADDBUFF Option	169
INDEXAREA Option	170
INDEXED Option	170
KEYLOC Option — Key Location	170
NOWRITE Option	172
Creating an Indexed Data Set	172
Essential Information	172
Name of the Data Set	175
Record Format and Keys	175
Overflow Area	177
Master Index	178
Accessing and Updating an Indexed Data Set	179
Using Sequential Access	180
Using Direct Access	181
Essential Information	181
Example	182
Reorganizing an Indexed Data Set	184

<b>Chapter 10. Defining and Using Regional Data Sets</b>	185
Defining Files for a Regional Data Set	188
Specifying ENVIRONMENT Options	188
REGIONAL Option	188
Using Keys with REGIONAL Data Sets	190
Using REGIONAL(1) Data Sets	190
Dummy Records	190
Creating a REGIONAL(1) Data Set	191
Example	191
Accessing and Updating a REGIONAL(1) Data Set	192
Sequential Access	193
Direct Access	193
Example	193
Using REGIONAL(2) Data Sets	195
Using Keys for REGIONAL(2) and (3) Data Sets	195
Dummy Records	196
Creating a REGIONAL(2) Data Set	197
Example	197
Accessing and Updating a REGIONAL(2) Data Set	198
Sequential Access	199
Direct Access	199
Example	199
Using REGIONAL(3) Data Sets	202
Dummy Records	202
Creating a REGIONAL(3) Data Set	202
Example	203
Accessing and Updating a REGIONAL(3) Data Set	204
Sequential Access	204
Direct Access	205
Example	205
Essential Information for Creating and Accessing Regional Data Sets	208
<b>Chapter 11. Defining and Using VSAM Data Sets</b>	211
Using VSAM Data Sets	211
How to Run a Program with VSAM Data Sets	211
Pairing an Alternate Index Path with a File	211
VSAM Organization	212
Keys for VSAM Data Sets	215
Keys for Indexed VSAM Data Sets	216
Relative Byte Addresses (RBA)	216
Relative Record Numbers	216
Choosing a Data Set Type	216
Defining Files for VSAM Data Sets	218
Specifying ENVIRONMENT Options	219
BKWD Option	220
BUFND Option	220
BUFNI Option	220
BUFSP Option	221
GENKEY Option	221
PASSWORD Option	221
REUSE Option	221
SIS Option	222
SKIP Option	222
VSAM Option	223

Performance Options	223
Defining Files for Alternate Index Paths	223
Using Files Defined for non-VSAM Data Sets	224
CONSECUTIVE Files	224
INDEXED Files	224
Using the VSAM Compatibility Interface	225
Adapting Existing Programs for VSAM	225
CONSECUTIVE Files	226
INDEXED Files	226
REGIONAL(1) Files	226
Using Several Files in One VSAM Data Set	226
Using Shared Data Sets	227
Defining VSAM Data Sets	227
Entry-Sequenced Data Sets	228
Loading an ESDS	229
Using a SEQUENTIAL File to Access an ESDS	229
Defining and Loading an ESDS	230
Updating an Entry-Sequenced Data Set	231
Key-Sequenced and Indexed Entry-Sequenced Data Sets	232
Loading a KSDS or Indexed ESDS	234
Using a SEQUENTIAL File to Access a KSDS or Indexed ESDS	236
Using a DIRECT File to Access a KSDS or Indexed ESDS	236
Alternate Indexes for KSDSs or Indexed ESDSs	239
Unique Key Alternate Index Path	239
Nonunique Key Alternate Index Path	240
Detecting Nonunique Alternate Index Keys	242
Using Alternate Indexes with ESDSs	242
Using Alternate Indexes with KSDSs	243
Relative-Record Data Sets	247
Loading an RRDS	249
Using a SEQUENTIAL File to Access an RRDS	251
Using a DIRECT File to Access an RRDS	252
<b>Chapter 12. Defining and Using Teleprocessing Data Sets</b>	<b>255</b>
Message Control Program (MCP)	255
TCAM Message Processing Program (TCAM MPP)	256
Teleprocessing Organization	256
Essential Information	257
Defining Files for a Teleprocessing Data Set	257
Specifying ENVIRONMENT Options	257
TP Option	257
RECSIZE Option	258
BUFFERS Option	258
Writing a TCAM Message Processing Program (TCAM MPP)	258
Handling PL/I Conditions	260
TCAM MPP Example	261

---

## Chapter 6. Using Data Sets and Files

Your PL/I programs process and transmit units of information called *records*. On MVS systems, a collection of records is called a *data set*. On VM, a collection of records is called a *file*. Data sets, and VM files, are physical collections of information external to PL/I programs; they can be created, accessed, or modified by programs written in PL/I or other languages or by the utility programs of the operating system.

Your PL/I program recognizes and processes information in a data set by using a symbolic or logical representation of the data set called a *file*. (Yes, in VM there are files defined within your program that are symbolic representations of files external to your program.) This chapter describes how to associate data sets or VM files with the files known within your program. It introduces the five major types of data sets, how they are organized and accessed, and some of the file and data set characteristics you need to know how to specify.

---

### Associating Data Sets with Files

A file used within a PL/I program has a *PL/I file name*. The physical data set external to the program has a name by which it is known to the operating system: under MVS or TSO it is a *data set name* or *dsname*, and on VM it is a *VM file name*. In some cases the data set or file has no name; it is known to the system by the device on which it exists.

The operating system needs a way to recognize which physical data set is referred to by your program, so you must provide a statement, external to your program, that associates the PL/I file name with a dsname or a VM file name:

- Under MVS batch, you must write a *data definition* or *DD* statement. For example, if you have the following file declaration in your program:

```
DCL STOCK FILE STREAM INPUT;
```

you should create a DD statement with a *data definition name* (*ddname*) that matches the name of the PL/I file. The DD statement specifies a physical data set name (*dsname*) and gives its characteristics:

```
//GO.STOCK DD DSN=PARTS.INSTOCK, . . .
```

You'll find some guidance in writing DD statements in this manual, but for more detail refer to the job control language (JCL) manuals for your system.

- Under TSO, you must write an *ALLOCATE* command. In the declaration shown above for the PL/I file *STOCK*, you should write a TSO *ALLOCATE* statement that associates the PL/I file name with the MVS data set name:

```
ALLOCATE FILE(STOCK) DATASET(PARTS.INSTOCK)
```

- Under VM, you must write a *FILEDEF* command. For the same *STOCK* file declaration, a VM *FILEDEF* should look something like this:

```
FILEDEF STOCK DISK INSTOCK PARTS fm
```

There is more than one way to associate a data set with a PL/I file. You associate a data set with a PL/I file by ensuring that the *ddname* of the DD statement that defines the data set is the same as *either*:

- The declared PL/I file name, *or*
- The character-string value of the expression specified in the TITLE option of the associated OPEN statement.

You must choose your PL/I file names so that the corresponding ddnames conform to the following restrictions:

- If a file is opened implicitly, or if no TITLE option is included in the OPEN statement that explicitly opens the file, the ddname defaults to the file name. If the file name is longer than 8 characters, the default ddname is composed of the first 8 characters of the file name.
- The character set of the job control language does not contain the break character (`_`). Consequently, this character cannot appear in ddnames. Do not use break characters among the first 8 characters of file names, unless the file is to be opened with a TITLE option with a valid ddname as its expression. The alphabetic extender characters `$`, `@`, and `#`, however, are valid for ddnames, but the first character must be one of the letters A through Z.

Since external names are limited to 7 characters, an external file name of more than 7 characters is shortened into a concatenation of the first 4 and the last 3 characters of the file name. Such a shortened name is *not*, however, the name used as the ddname in the associated DD statement.

Consider the following statements:

1. OPEN FILE(MASTER);
2. OPEN FILE(OLDMASTER);
3. READ FILE(DETAIL) ...;

When statement number 1 is run, the file name MASTER is taken to be the same as the ddname of a DD statement in the current job step. When statement number 2 is run, the name OLDMASTE is taken to be the same as the ddname of a DD statement in the current job step. (The first 8 characters of a file name form the ddname. If OLDMASTER is an external name, it will be shortened by the compiler to OLDMASTER for use within the program.) If statement number 3 causes implicit opening of the file DETAIL, the name DETAIL is taken to be the same as the ddname of a DD statement in the current job step.

In each of the above cases, a corresponding DD statement or an equivalent TSO allocate or VM FILEDEF must appear in the job stream; otherwise, the UNDEFINEDFILE condition is raised. The three DD statements could start as follows:

1. //MASTER DD ...
2. //OLDMASTE DD ...
3. //DETAIL DD ...

If the file reference in the statement which explicitly or implicitly opens the file is not a file constant, the DD statement name must be the same as the *value* of the file reference. The following example illustrates how a DD statement should be associated with the value of a file variable:

```

DCL PRICES FILE VARIABLE,
RPRICE FILE;
PRICES = RPRICE;
OPEN FILE(PRICES);

```

The DD statement should associate the data set with the file constant RPRICE, which is the value of the file variable PRICES, thus:

```
//RPRICE DD DSNAME=...
```

Use of a file variable also allows you to manipulate a number of files at various times by a single statement. For example:

```

DECLARE F FILE VARIABLE,
A FILE,
B FILE,
C FILE;
.
.
.
DO F=A,B,C;
READ FILE (F) ...;
.
.
.
END;

```

The READ statement reads the three files A, B, and C, each of which can be associated with a different data set. The files A, B, and C remain open after the READ statement is executed in each instance.

The following OPEN statement illustrates use of the TITLE option:

```
OPEN FILE(DETAIL) TITLE('DETAIL1');
```

For this statement to be executed successfully, you must have a DD statement in the current job step with DETAIL1 as its ddname. It could start as follows:

```
//DETAIL1 DD DSNAME=DETAILA,...
```

Thus, you associate the data set DETAILA with the file DETAIL through the ddname DETAIL1.

## Associating Several Files with One Data Set

You can use the TITLE option to associate two or more PL/I files with the same external data set at the same time. This is illustrated in the following example, where INVNTY is the name of a DD statement defining a data set to be associated with two files:

```

OPEN FILE (FILE1) TITLE('INVNTY');
OPEN FILE (FILE2) TITLE('INVNTY');

```

If you do this, be careful. These two files access a common data set through separate control blocks and data buffers. When records are written to the data set from one file, the control information for the second file will not record that fact. Records written from the second file could then destroy records written from the first file. PL/I does not protect against data set damage that might occur. If the data set is extended, the extension is reflected only in the control blocks associated with the file that wrote the data; this can cause an abend when other files access the data set.

## Associating Several Data Sets with One File

The file name can, at different times, represent entirely different data sets. In the above example of the OPEN statement, the file DETAIL1 is associated with the data set named in the DSNNAME parameter of the DD statement DETAIL1. If you closed and reopened the file, you could specify a different ddname in the TITLE option to associate the file with a different data set.

Use of the TITLE option allows you to choose dynamically, at open time, one among several data sets to be associated with a particular file name. Consider the following example:

```
DO IDENT='A','B','C';
  OPEN FILE(MASTER)
    TITLE('MASTER1' || IDENT);
  .
  .
  .
  CLOSE FILE(MASTER);
END;
```

In this example, when MASTER is opened during the first iteration of the do-group, the associated ddname is taken to be MASTER1A. After processing, the file is closed, dissociating the file name and the ddname. During the second iteration of the do-group, MASTER is opened again. This time, MASTER is associated with the ddname MASTER1B. Similarly, during the final iteration of the do-group, MASTER is associated with the ddname MASTER1C.

## Concatenating Several Data Sets

Under MVS, for input only, you can concatenate two or more sequential or regional data sets (that is, link them so that they are processed as one continuous data set) by omitting the ddname from all but the first of the DD statements that describe them. For example, the following DD statements cause the data sets LIST1, LIST2, and LIST3 to be treated as a single data set for the duration of the job step in which the statements appear:

```
//GO.LIST DD DSNNAME=LIST1,DISP=OLD
//          DD DSNNAME=LIST2,DISP=OLD
//          DD DSNNAME=LIST3,DISP=OLD
```

When read from a PL/I program, the concatenated data sets need not be on the same volume. You cannot process concatenated data sets backward.

---

## Establishing Data Set Characteristics

A data set consists of records stored in a particular format which the operating system data management routines understand. When you declare or open a file in your program, you are describing to PL/I and to the operating system the characteristics of the records that file will contain. You can also use JCL, TSO ALLOCATEs, or CMS FILEDEFs, to describe to the operating system the characteristics of the data in data sets or in the PL/I files associated with them.

You do not always need to describe your data both within the program and outside it; often one description will serve for both data sets and their associated PL/I files. There are, in fact, advantages to describing your data's characteristics in only one place. These are described later in this chapter and in following chapters.

To effectively describe your program data and the data sets you will be using, you need to understand something of how the operating system moves and stores data.

## Blocks and Records

The items of data in a data set are arranged in blocks separated by interblock gaps (IBG). (Some manuals refer to these as interrecord gaps.)

A *block* is the unit of data transmitted to and from a data set. Each block contains one record, part of a record, or several records. You can specify the block size in the BLKSIZE parameter of the DD, ALLOCATE, or FILEDEF statement or in the BLKSIZE option of the ENVIRONMENT attribute.

A *record* is the unit of data transmitted to and from a program. You can specify the record length in the LRECL parameter of the DD, ALLOCATE, or FILEDEF statement or in the RECSIZE option of the ENVIRONMENT attribute.

When writing a PL/I program, you need consider only the records that you are reading or writing; but when you describe the data sets that your program will create or access, you must be aware of the relationship between blocks and records.

Blocking conserves storage space in a magnetic storage volume because it reduces the number of interblock gaps, and it can increase efficiency by reducing the number of input/output operations required to process a data set. Records are blocked and deblocked by the data management routines.

**Information Interchange Codes:** The normal code in which data is recorded is the Extended Binary Coded Decimal Interchange Code (EBCDIC). However, for magnetic tape only, the operating system accepts data recorded in the American Standard Code for Information Interchange (ASCII). You use the ASCII and BUFOFF options of the ENVIRONMENT attribute if your program will read or write data sets recorded in ASCII.

A prefix field up to 99 bytes in length might be present at the beginning of each block in an ASCII data set. The use of this field is controlled by the BUFOFF option of the ENVIRONMENT attribute. For a full description of the ASCII option, see "ASCII" on page 154.

Each character in the ASCII code is represented by a 7-bit pattern and there are 128 such patterns. The ASCII set includes a substitute character (the SUB control character) that is used to represent EBCDIC characters having no valid ASCII code. The ASCII substitute character is translated to the EBCDIC SUB character, which has the bit pattern 00111111.

## Record Formats

The records in a data set have one of the following formats:

- Fixed-length
- Variable-length
- Undefined-length.

Records can be blocked if required. The operating system will deblock fixed-length and variable-length records, but you must provide code in your program to deblock undefined-length records.

You specify the record format in the RECFM parameter of the DD, ALLOCATE, or FILEDEF statement or as an option of the ENVIRONMENT attribute.

## Fixed-Length Records

You can specify the following formats for fixed-length records:

- F Fixed-length, unblocked
- FB Fixed-length, blocked
- FS Fixed-length, unblocked, standard
- FBS Fixed-length, blocked, standard.

In a data set with fixed-length records, as shown in Figure 20, all records have the same length. If the records are blocked, each block usually contains an equal number of fixed-length records (although a block can be truncated). If the records are unblocked, each record constitutes a block.

---

### Unblocked records (F-format):



### Blocked records (FB-format):



---

Figure 20. Fixed-Length Records

Because it bases blocking and deblocking on a constant record length, the operating system processes fixed-length records faster than variable-length records.

The use of “standard” (FS-format and FBS-format) records further optimizes the sequential processing of a data set on a direct-access device. A standard format data set must contain fixed-length records and must have no embedded empty tracks or short blocks (apart from the last block). With a standard format data set, the operating system can predict whether the next block of data will be on a new track and, if necessary, can select a new read/write head in anticipation of the transmission of that block. A PL/I program never places embedded short blocks in a data set with fixed-length records. A data set containing fixed-length records can be processed as a standard data set even if it is not created as such, providing it contains no embedded short blocks or empty tracks.

## Variable-Length Records

You can specify the following formats for variable-length records:

- V Variable-length, unblocked
- VB Variable-length, blocked
- VS Variable-length, unblocked, spanned
- VBS Variable-length, blocked, spanned
- D Variable-length, unblocked, ASCII
- DB Variable-length, blocked, ASCII.

V-format allows both variable-length records and variable-length blocks. A 4-byte prefix of each record and the first 4 bytes of each block contain control information for use by the operating system (including the length in bytes of the record or

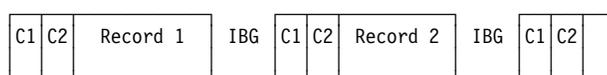
block). Because of these control fields, variable-length records cannot be read backward. Illustrations of variable-length records are shown in Figure 21 on page 105.

V-format signifies unblocked variable-length records. Each record is treated as a block containing only one record. The first 4 bytes of the block contain block control information, and the next 4 contain record control information.

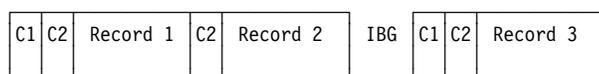
VB-format signifies blocked variable-length records. Each block contains as many complete records as it can accommodate. The first 4 bytes of the block contain block control information, and a 4-byte prefix of each record contains record control information.

---

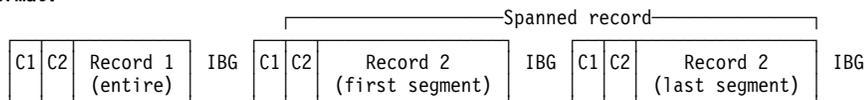
**V-format:**



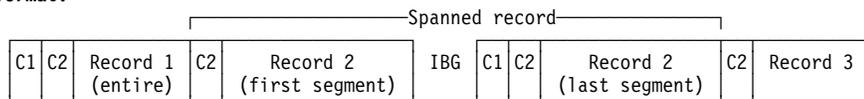
**VB-format:**



**VS-format:**



**VBS-format:**



C1: Block control information  
 C2: Record or segment control information

---

Figure 21. Variable-Length Records

**Spanned Records:** A spanned record is a variable-length record in which the length of the record can exceed the size of a block. If this occurs, the record is divided into segments and accommodated in two or more consecutive blocks by specifying the record format as either VS or VBS. Segmentation and reassembly are handled by the operating system. The use of spanned records allows you to select a block size, independently of record length, that will combine optimum use of auxiliary storage with maximum efficiency of transmission.

VS-format is similar to V-format. Each block contains only one record or segment of a record. The first 4 bytes of the block contain block control information, and the next 4 contain record or segment control information (including an indication of whether the record is complete or is a first, intermediate, or last segment).

With REGIONAL(3) organization, the use of VS-format removes the limitations on block size imposed by the physical characteristics of the direct-access device. If

the record length exceeds the size of a track, or if there is no room left on the current track for the record, the record will be spanned over one or more tracks.

VBS-format differs from VS-format in that each block contains as many complete records or segments as it can accommodate; each block is, therefore, approximately the same size (although there can be a variation of up to 4 bytes, since each segment must contain at least 1 byte of data).

**ASCII Records:** For data sets that are recorded in ASCII, use D-format as follows:

- D-format records are similar to V-format, except that the data they contain is recorded in ASCII.
- DB-format records are similar to VB-format, except that the data they contain is recorded in ASCII.

### Undefined-Length Records

U-format allows the processing of records that do not conform to F- and V-formats. The operating system and the compiler treat each block as a record; your program must perform any required blocking or deblocking.

## Data Set Organization

The data management routines of the operating system can handle a number of types of data sets, which differ in the way data is stored within them and in the allowed means of access to the data. The three main types of non-VSAM data sets and the corresponding keywords describing their PL/I organization<sup>1</sup> are as follows:

Type of data set	PL/I organization
Sequential	CONSECUTIVE
Indexed sequential	INDEXED
Direct	REGIONAL

The compiler recognizes a fourth type, *teleprocessing*, by the file attribute TRANSIENT.

A fifth type, *partitioned*, has no corresponding PL/I organization.

PL/I also provides support for three types of VSAM data organization: *ESDS*, *KSDS*, and *RRDS*. For more information about VSAM data sets, see Chapter 11, "Defining and Using VSAM Data Sets" on page 211.

In a *sequential* (or CONSECUTIVE) data set, records are placed in physical sequence. Given one record, the location of the next record is determined by its physical position in the data set. Sequential organization is used for all magnetic tapes, and can be selected for direct-access devices.

An *indexed sequential* (or INDEXED) data set must reside on a direct-access volume. An index or set of indexes maintained by the operating system gives the

---

<sup>1</sup> Do not confuse the terms "sequential" and "direct" with the PL/I file attributes SEQUENTIAL and DIRECT. The attributes refer to how the file is to be processed, and not to the way the corresponding data set is organized.

location of certain principal records. This allows direct retrieval, replacement, addition, and deletion of records, as well as sequential processing.

A *direct* (or REGIONAL) data set must reside on a direct-access volume. The records within the data set can be organized in three ways: REGIONAL(1), REGIONAL(2), and REGIONAL(3); in each case, the data set is divided into regions, each of which contains one or more records. A key that specifies the region number and, for REGIONAL(2) and REGIONAL(3), identifies the record, allows direct-access to any record; sequential processing is also possible.

A *teleprocessing* data set (associated with a TRANSIENT file in a PL/I program) must reside in storage. Records are placed in physical sequence.

In a *partitioned* data set, independent groups of sequentially organized data, each called a member, reside in a direct-access data set. The data set includes a directory that lists the location of each member. Partitioned data sets are often called *libraries*. The compiler includes no special facilities for creating and accessing partitioned data sets. Each member can be processed as a CONSECUTIVE data set by a PL/I program. The use of partitioned data sets as libraries is described under Chapter 7, "Using Libraries" on page 123.

## Labels

The operating system uses internal labels to identify magnetic-tape and direct-access volumes, and to store data set attributes (for example, record length and block size). The attribute information must originally come from a DD statement or from your program.

Magnetic-tape volumes can have IBM standard or nonstandard labels, or they can be unlabeled. IBM standard labels have two parts: the initial volume label, and header and trailer labels. The initial volume label identifies a volume and its owner; the header and trailer labels precede and follow each data set on the volume. Header labels contain system information, device-dependent information (for example, recording technique), and data-set characteristics. Trailer labels are almost identical with header labels, and are used when magnetic tape is read backward.

Direct-access volumes have IBM standard labels. Each volume is identified by a volume label, which is stored on the volume. This label contains a volume serial number and the address of a volume table of contents (VTOC). The table of contents, in turn, contains a label, termed a *data set control block* (DSCB), for each data set stored on the volume.

## Data Definition (DD) Statement

A data definition (DD) statement is a job control statement that defines a data set to the operating system, and is a request to the operating system for the allocation of input/output resources. If the data sets are not dynamically allocated, each job step must include a DD statement for each data set that is processed by the step.

Your *MVS/ESA JCL User's Guide* describes the syntax of job control statements. The operand field of the DD statement can contain keyword parameters that describe the location of the data set (for example, volume serial number and identification of the unit on which the volume will be mounted) and the attributes of the data itself (for example, record format).

The DD statement enables you to write PL/I source programs that are independent of the data sets and input/output devices they will use. You can modify the parameters of a data set or process different data sets without recompiling your program.

The following paragraphs describe the relationship of some operands of the DD statement to your PL/I program.

The LEAVE and REREAD options of the ENVIRONMENT attribute allow you to use the DISP parameter to control the action taken when the end of a magnetic-tape volume is reached or when a magnetic-tape data set is closed. The LEAVE and REREAD options are described under “LEAVE|REREAD” on page 153, and are also described under “CLOSE Statement” in *PL/I for MVS & VM Language Reference*.

Write validity checking, which was standard in PL/I Version 1, is no longer performed. Write validity checking can be requested through the OPTCD subparameter of the DCB parameter of the JCL DD statement. See the *OS/VS2 TSO Command Language Reference* and *OS/VS2 Job Control Language* manuals.

### **Use of the Conditional Subparameters**

If you use the conditional subparameters of the DISP parameter for data sets processed by PL/I programs, the step abend facility must be used. The step abend facility is obtained as follows:

1. The ERROR condition should be raised or signaled whenever the program is to terminate execution after a failure that requires the application of the conditional subparameters.
2. The PL/I user exit must be changed to request an ABEND.

### **Data Set Characteristics**

The DCB (data control block) parameter of the DD statement allows you to describe the characteristics of the data in a data set, and the way it will be processed, at run time. Whereas the other parameters of the DD statement deal chiefly with the identity, location, and disposal of the data set, the DCB parameter specifies information required for the processing of the records themselves. The subparameters of the DCB parameter are described in your *MVS/ESA JCL User's Guide*.

The DCB parameter contains subparameters that describe:

- The organization of the data set and how it will be accessed (CYLOFL, DSORG, LIMCT, NCP, NTM, and OPTCD subparameters)
- Device-dependent information such as the recording technique for magnetic tape or the line spacing for a printer (CODE, DEN, FUNC, MODE, OPTCD=J, PRTSP, STACK, and TRTCH subparameters)
- The record format (BLKSIZE, KEYLEN, LRECL, RECFM, and RKP subparameters)
- The number of buffers that are to be used (BUFNO subparameter)
- The ASA control characters (if any) that will be inserted in the first byte of each record (RECFM subparameter).

You can specify BLKSIZE, BUFNO, LRECL, KEYLEN, NCP, RECFM, RKP, and TRKOFL (or their equivalents) in the ENVIRONMENT attribute of a file declaration in your PL/I program instead of in the DCB parameter.

You cannot use the DCB parameter to override information already established for the data set in your PL/I program (by the file attributes declared and the other attributes that are implied by them). DCB subparameters that attempt to change information already supplied are ignored.

An example of the DCB parameter is:

```
DCB=(RECFM=FB,BLKSIZE=400,LRECL=40)
```

which specifies that fixed-length records, 40 bytes in length, are to be grouped together in a block 400 bytes long.

## Associating PL/I Files with Data Sets

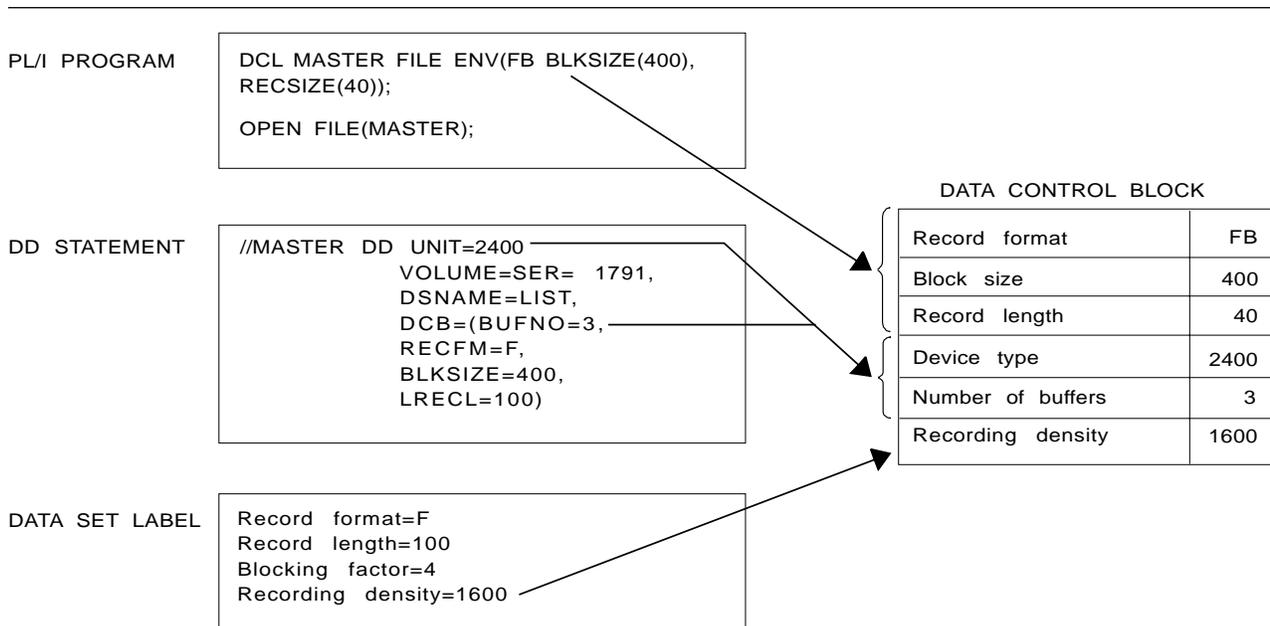
**Opening a File:** The execution of a PL/I OPEN statement associates a file with a data set. This requires merging of the information describing the file and the data set. If any conflict is detected between file attributes and data set characteristics, the UNDEFINEDFILE condition is raised.

Subroutines of the PL/I library create a skeleton data control block for the data set. They use the file attributes from the DECLARE and OPEN statements and any attributes implied by the declared attributes, to complete the data control block as far as possible. (See Figure 22 on page 110.) They then issue an OPEN macro instruction, which calls the data management routines to check that the correct volume is mounted and to complete the data control block.

The data management routines examine the data control block to see what information is still needed and then look for this information, first in the DD statement, and finally, if the data set exists and has standard labels, in the data set labels. For new data sets, the data management routines begin to create the labels (if they are required) and to fill them with information from the data control block.

Neither the DD statement nor the data set label can override information provided by the PL/I program; nor can the data set label override information provided by the DD statement.

When the DCB fields are filled in from these sources, control returns to the PL/I library subroutines. If any fields still are not filled in, the PL/I OPEN subroutine provides default information for some of them. For example, if LRECL is not specified, it is provided from the value given for BLKSIZE.



Note: Information from the PL/I program overrides that from the DD statement and the data set label. Information from the DD statement overrides that from the data set label.

Figure 22. How the Operating System Completes the DCB

**Closing a File:** The execution of a PL/I CLOSE statement dissociates a file from the data set with which it was associated. The PL/I library subroutines first issue a CLOSE macro instruction and, when control returns from the data management routines, release the data control block that was created when the file was opened. The data management routines complete the writing of labels for new data sets and update the labels of existing data sets.

## Specifying Characteristics in the ENVIRONMENT Attribute

You can use various options in the ENVIRONMENT attribute. Each type of file has different attributes and environment options, which are listed below.

**The ENVIRONMENT Attribute:** You use the ENVIRONMENT attribute of a PL/I file declaration file to specify information about the physical organization of the data set associated with a file, and other related information. The format of this information must be a parenthesized option list.

►—ENVIRONMENT—(—option-list—)—————►

Abbreviation: ENV

You can specify the options in any order, separated by blanks or commas.

The following example illustrates the syntax of the ENVIRONMENT attribute in the context of a complete file declaration (the options specified are for VSAM and are discussed in Chapter 11, "Defining and Using VSAM Data Sets" on page 211).

DCL FILENAME FILE RECORD SEQUENTIAL  
INPUT ENV(VSAM GENKEY);

Table 15 summarizes the ENVIRONMENT options and file attributes. Certain qualifications on their use are presented in the notes and comments for the figure. Those options that apply to more than one data set organization are described in the remainder of this chapter. In addition, in the following chapters, each option is described with each data set organization to which it applies.

Table 15 (Page 1 of 2). Attributes of PL/I File Declarations

Data set type	Stream	Record										Legend: C Checked for VSAM D Default I Must be specified or implied N Ignored for VSAM O Optional S Must be specified - Invalid
		Sequential					Direct					
		Consecutive		Regional		Te le pr oc e s s i n g	I n d e x e d	V S A M	R e g i o n a l	I n d e x e d	V S A M	
B u f f e r e d	U n b u f f e r e d	B u f f e r e d	U n b u f f e r e d									
File Type	C o n s e c u t i v e	B u f f e r e d	U n b u f f e r e d	B u f f e r e d	U n b u f f e r e d	T e l e p r o c e s s i n g	I n d e x e d	V S A M	R e g i o n a l	I n d e x e d	V S A M	
<b>File attributes<sup>1</sup></b>												<b>Attributes implied</b>
File	I	I	I	I	I	I	I	I	I	I	I	File
Input <sup>1</sup>	D	D	D	D	D	D	D	D	D	D	D	File
Output	O	O	O	O	O	O	O	O	O	O	O	File
Environment	I	I	I	S	S	S	S	S	S	S	S	File
Stream	D	-	-	-	-	-	-	-	-	-	-	File
Print <sup>1</sup>	O	-	-	-	-	-	-	-	-	-	-	File stream output
Record	-	I	I	I	I	I	I	I	I	I	I	File
Update <sup>2</sup>	-	O	O	O	O	-	O	O	O	O	O	File record
Sequential	-	D	D	D	D	-	D	D	-	-	D	File record
Buffered	-	D	-	D	-	I	D	D	-	-	S	File record
Unbuffered	-	-	S	-	S	-	-	S	D	D	D	File record
Backwards <sup>3</sup>	-	O	O	-	-	-	-	-	-	-	-	File record sequential input
Transient	-	-	-	-	-	I	-	-	-	-	-	File
Keyed <sup>4</sup>	-	-	-	O	O	I	O	O	I	I	O	File record
Direct	-	-	-	-	-	-	-	S	S	S	S	File record keyed
Exclusive	-	-	-	-	-	-	-	-	O	O	-	File record direct keyed update
<b>ENVIRONMENT options</b>												<b>Comments</b>
F FB FS FBS V  VB VS VBS U	I	S	S	-	-	-	-	N	-	-	N	VS and VBS are invalid with STREAM
F FB D DB U	S	S	-	-	-	-	-	N	-	-	N	ASCII data sets only
F V VS U	-	-	-	S	S	-	-	N	S	-	N	Only F for REGIONAL(1) and (2)
F FB V VB	-	-	-	-	-	-	S	N	-	S	N	VS invalid with UNBUF
RECSIZE(n)	I	I	I	I	I	S	I	C	I	I	C	RECSIZE and/or BLKSIZE must be specified
BLKSIZE(n)	I	I	I	I	I	-	I	N	I	I	N	for consecutive, indexed, and regional files
NCP(n)	-	O	O	O	O	-	O	N	O	O	N	NCP>1 for VSAM specifies ISAM compatibility
TRKOFI	-	O	O	O	O	-	-	-	O	-	-	Invalid for REGIONAL(3)
KEYLENGTH(n)	-	-	-	S	S	-	S	C	S	S	C	For REGIONAL(2) and (3) OUTPUT ONLY
COBOL	-	O	O	O	O	-	O	O	O	O	O	
BUFFERS(n)	I	I	-	I	-	I	I	N	-	-	N	
SCALARVARYING	-	O	O	O	O	-	O	O	O	O	O	Invalid for ASCII data sets
CONSECUTIVE	D	D	D	-	-	-	-	O	-	-	O	Allowed for VSAM ESDS
TOTAL	-	O	O	-	-	-	-	-	-	-	-	
LEAVE	O	O	O	-	-	-	-	-	-	-	-	
REREAD	O	O	O	-	-	-	-	-	-	-	-	
ASCII	O	O	-	-	-	-	-	-	-	-	-	
BUFOFF(n)	O	O	-	-	-	-	-	-	-	-	-	
CTLASA CTL360	-	O	O	-	-	-	-	-	-	-	-	Invalid for ASCII data sets
GRAPHIC	O	-	-	-	-	-	-	-	-	-	-	
TP (M R)	-	-	-	-	-	S	-	-	-	-	-	
INDEXED	-	-	-	-	-	-	S	O	-	S	O	Allowed for VSAM ESDS
KEYLOC(n)	-	-	-	-	-	-	O	-	-	O	-	
INDEXAREA(n)	-	-	-	-	-	-	-	-	-	O	-	
ADDBUFF	-	-	-	-	-	-	-	-	-	O	-	
NOWRITE	-	-	-	-	-	-	-	-	-	O	-	UPDATE files only
GENKEY	-	-	-	-	-	-	O	O	-	O	O	INPUT or UPDATE files only; KEYED is required
REGIONAL((1 2 3))	-	-	-	S	S	-	-	-	S	-	-	
VSAM	-	-	-	-	-	-	-	S	-	-	S	

Table 15 (Page 2 of 2). Attributes of PL/I File Declarations

Data set type	Stream	Record										Legend: C Checked for VSAM D Default I Must be specified or implied N Ignored for VSAM O Optional S Must be specified - Invalid
		Sequential						Direct				
		Consecutive		Regional		Te le pr oc e s s i n g	I n d e x e d	V S A M	R e g i o n a l	I n d e x e d	V S A M	
B u f f e r e d	U n b u f f e r e d	B u f f e r e d	U n b u f f e r e d									
PASSWORD	-	-	-	-	-	-	-	O	-	-	O	OUTPUT file only
SIS	-	-	-	-	-	-	-	-	-	-	O	
SKIP	-	-	-	-	-	-	-	O	-	-	-	
BKWD	-	-	-	-	-	-	-	O	-	-	O	
REUSE	-	-	-	-	-	-	-	O	-	-	O	
BUFND(n)	-	-	-	-	-	-	-	O	-	-	O	
BUFNI(n)	-	-	-	-	-	-	-	O	-	-	O	
BUFSP(n)	-	-	-	-	-	-	-	O	-	-	O	

Notes:

1. A file with the INPUT attribute cannot have the PRINT attribute.
2. UPDATE is invalid for tape files.
3. BACKWARDS is valid only for input tape files.
4. Keyed is required for INDEXED and REGIONAL output.

**Data Set Organization Options:** The options that specify data set organization are:



Each option is described in the discussion of the data set organization to which it applies.

If you don't specify the data set organization option in the ENVIRONMENT attribute, it defaults in the following manner when the file is opened:

- If merged attributes from DECLARE and OPEN statements do not include TRANSIENT, the default is CONSECUTIVE.
- If the attributes include TRANSIENT, the default is TP(M).

**Other ENVIRONMENT Options:** You can use a constant or variable with those ENVIRONMENT options that require integer arguments, such as block sizes and record lengths. The variable must not be subscripted or qualified, and must have attributes FIXED BINARY(31,0) and STATIC.

Some of the information that can be specified in the options of the ENVIRONMENT attribute can also be specified—when TOTAL is not specified—in the subparameters of the DCB parameter of a DD statement. The list of equivalents for ENVIRONMENT options and DCB parameters are:

**ENVIRONMENT option    DCB subparameter**

Record format	RECFM <sup>1</sup>
RECSIZE	LRECL
BLKSIZE	BLKSIZE
BUFFERS	BUFNO
CTLASA CTL360	RECFM
NCP	NCP
TRKOFL	RECFM
KEYLENGTH	KEYLEN
KEYLOC	RKP
ASCII	ASCII
BUFOFF	BUFOFF

**Note:** <sup>1</sup>VS must be specified as an ENVIRONMENT option, not in the DCB.

**Record Formats for Record-Oriented Data Transmission:** Record formats supported depend on the data set organization.



Records can have one of the following formats:

Fixed-length	F	unblocked
	FB	blocked
	FS	unblocked, standard
	FBS	blocked, standard
Variable-length	V	unblocked
	VB	blocked
	VS	spanned
	VBS	blocked, spanned
	D	unblocked, ASCII
	DB	blocked, ASCII
Undefined-length	U	(cannot be blocked)

When U-format records are read into a varying-length string, PL/I sets the length of the string to the block length of the retrieved data.

These record format options do not apply to VSAM data sets. If you specify a record format option for a file associated with a VSAM data set, the option is ignored.

You can only specify VS-format records for data sets with consecutive or REGIONAL(3) organization.

**Record Formats for Stream-Oriented Data Transmission:** The record format options for stream-oriented data transmission are discussed in “Using Stream-Oriented Data Transmission” on page 129.

**RECSIZE Option:** The RECSIZE option specifies the record length.

►—RECSIZE—(—*record-length*—)—————►

For files other than TRANSIENT files and files associated with VSAM data sets, **record-length** is the sum of:

1. The length required for data. For variable-length and undefined-length records, this is the maximum length.
2. Any control bytes required. Variable-length records require 4 (for the record-length prefix); fixed-length and undefined-length records do not require any.

For a TRANSIENT file, it is the sum of:

1. The four V-format control bytes
2. One flag byte
3. Eight bytes for the key (origin or destination identifier)
4. The maximum length required for the data.

For VSAM data sets, the maximum and average lengths of the records are specified to the Access Method Services utility when the data set is defined. If you include the RECSIZE option in the file declaration for checking purposes, you should specify the maximum record size. If you specify RECSIZE and it conflicts with the values defined for the data set, the UNDEFINEDFILE condition is raised.

You can specify **record-length** as an integer or as a variable with attributes FIXED BINARY(31,0) STATIC.

The value is subject to the following conventions:

*Maximum:*

Fixed-length, and undefined (except ASCII data sets): 32760

V-format, and VS- and VBS-format with UPDATE files: 32756

VS- and VBS-format with INPUT and OUTPUT files: 16777215

ASCII data sets: 9999

VSAM data sets: 32761 for unspanned records. For spanned records, the maximum is the size of the control area.

**Note:** For VS- and VBS-format records longer than 32,756 bytes, you must specify the length in the RECSIZE option of ENVIRONMENT, and for the DCB subparameter of the DD statement you must specify LRECL=X. If RECSIZE exceeds the allowed maximum for INPUT or OUTPUT, either a record condition occurs or the record is truncated.

*Zero value:*

A search for a valid value is made *first*:

- In the DD statement for the data set associated with the file, and *second*
- In the data set label.

If neither of these provides a value, default action is taken (see “Record Format, BLKSIZE, and RECSIZE Defaults” on page 117).

*Negative Value:*

The UNDEFINEDFILE condition is raised.

**BLKSIZE Option:** The BLKSIZE option specifies the maximum block size on the data set.

▶—BLKSIZE—(—*block-size*—)—————▶

**block-size** is the sum of:

1. The total length(s) of one of the following:

- A single record
- A single record and either one or two record segments
- Several records
- Several records and either one or two record segments
- Two record segments
- A single record segment.

For variable-length records, the length of each record or record segment includes the 4 control bytes for the record or segment length.

The above list summarizes all the possible combinations of records and record segments options: fixed- or variable-length blocked or unblocked, spanned or unspanned. When specifying a block size for spanned records, you must be aware that each record and each record segment requires 4 control bytes for the record length, and that these quantities are in addition to the 4 control bytes required for each block.

2. Any further control bytes required.
  - Variable-length blocked records require 4 (for the block size).
  - Fixed-length and undefined-length records do not require any further control bytes.
3. Any block prefix bytes required (ASCII data sets only).

**block-size** can be specified as an integer, or as a variable with attributes FIXED BINARY(31,0) STATIC.

The value is subject to the following conventions:

*Maximum:*

32760 (or 9999 for an ASCII data set for which BUFOFF without a prefix-length value has been specified).

In regional 3 files, the maximum declared block size must not exceed 32,680 bytes. This is because the 32,760 byte maximum for block size consists of the declared block size plus the key length plus the length of the IOCB. If you declare "BLKSIZE=32760", when the keylength and IOCB length are added to it, the maximum is exceeded and an "UNDEFINED FILE" error message is issued.

*Zero value:*

If you set BLKSIZE to 0, under MVS, the Data Facility Product sets the block size. For an elaboration of this topic, see "Record Format, BLKSIZE, and RECSIZE Defaults" on page 117. BLKSIZE defaults.

*Negative value:*

The UNDEFINEDFILE condition is raised.

The relationship of block size to record length depends on the record format:

*FB-format or FBS-format:*

The block size must be a multiple of the record length.

*VB-format:*

The block size must be equal to or greater than the sum of:

1. The maximum length of any record
2. Four control bytes.

*VS-format or VBS-format:*

The block size can be less than, equal to, or greater than the record length.

*DB-format:*

The block size must be equal to or greater than the sum of:

1. The maximum length of any record
2. The length of the block prefix (if block is prefixed).

**Notes:**

- Use the BLKSIZE option with unblocked (F-, V-, or D-format) records in either of the following ways:
  - Specify the BLKSIZE option, but not the RECSIZE option. Set the record length equal to the block size (minus any control or prefix bytes), and leave the record format unchanged.

- Specify both BLKSIZE and RECSIZE and ensure that the relationship of the two values is compatible with blocking for the record format you use. Set the record format to FB, VB, or DB, whichever is appropriate.
- If for FB-format or FBS-format records the block size equals the record length, the record format is set to F.
- For REGIONAL(3) data sets with VS format, record length cannot be greater than block size.
- The BLKSIZE option does not apply to VSAM data sets, and is ignored if you specify it for one.

**Record Format, BLKSIZE, and RECSIZE Defaults:** If you do not specify either the record format, block size, or record length for a non-VSAM data set, the following default action is taken:

*Record format:*

A search is made in the associated DD statement or data set label. If the search does not provide a value, the UNDEFINEDFILE condition is raised, except for files associated with dummy data sets or the foreground terminal, in which case the record format is set to U.

*Block size or record length:*

If one of these is specified, a search is made for the other in the associated DD statement or data set label. If the search provides a value, and if this value is incompatible with the value in the specified option, the UNDEFINEDFILE condition is raised. If the search is unsuccessful, a value is derived from the specified option (with the addition or subtraction of any control or prefix bytes).

If neither is specified, the UNDEFINEDFILE condition is raised, except for files associated with dummy data sets, in which case BLKSIZE is set to 121 for F-format or U-format records and to 129 for V-format records. For files associated with the foreground terminal, RECSIZE is set to 120.

If you are using MVS with the Data Facility Product system-determined block size, DFP determines the optimum block size for the device type assigned. If you specify BLKSIZE(0) in either the DD assignment or the ENVIRONMENT statement, DFP calculates BLKSIZE using the record length, record format, and device type.

**BUFFERS Option:** A buffer is a storage area that is used for the intermediate storage of data transmitted to and from a data set. The use of buffers can speed up processing of SEQUENTIAL files. Buffers are essential for blocking and deblocking records and for locate-mode transmission.

Use the BUFFERS option in the ENVIRONMENT attribute to specify buffers to be allocated for CONSECUTIVE and INDEXED data sets, according to the following syntax:

►—BUFFERS—(—*n*—)—————►

where *n* is the number of buffers you want allocated for your data set, not to exceed 255 (or such other maximum as is established for your PL/I installation).

If you specify zero, PL/I uses two buffers. A REGIONAL data set is always allocated two buffers.

In teleprocessing, the BUFFERS option specifies the number of buffers available for a particular message queue; that is, for a particular TRANSIENT file. The buffer size is specified in the message control program for the installation. The number of buffers specified should, if possible, be sufficient to provide for the longest message to be transmitted.

The BUFFERS option is ignored for VSAM; you use the BUFNI, BUFND, and BUFSP options instead.

**GENKEY Option — Key Classification:** The GENKEY (generic key) option applies only to INDEXED and VSAM key-sequenced data sets. It enables you to classify keys recorded in a data set and to use a SEQUENTIAL KEYED INPUT or SEQUENTIAL KEYED UPDATE file to access records according to their key classes.

►—GENKEY—————►

A generic key is a character string that identifies a class of keys; all keys that begin with the string are members of that class. For example, the recorded keys “ABCD”, “ABCE”, and “ABDF” are all members of the classes identified by the generic keys “A” and “AB”, and the first two are also members of the class “ABC”; and the three recorded keys can be considered to be unique members of the classes “ABCD”, “ABCE”, and “ABDF”, respectively.

The GENKEY option allows you to start sequential reading or updating of a VSAM data set from the first record that has a key in a particular class, and for an INDEXED data set from the first nondummy record that has a key in a particular class. You identify the class by including its generic key in the KEY option of a READ statement. Subsequent records can be read by READ statements without the KEY option. No indication is given when the end of a key class is reached.

Although you can retrieve the first record having a key in a particular class by using a READ with the KEY option, you cannot obtain the actual key unless the records have embedded keys, since the KEYTO option cannot be used in the same statement as the KEY option.

In the following example, a key length of more than 3 bytes is assumed:

```
DCL IND FILE RECORD SEQUENTIAL KEYED
  UPDATE ENV (INDEXED GENKEY);
.
.
.
  READ FILE(IND) INTO(INFIELD)
    KEY ('ABC');
.
.
.
NEXT: READ FILE (IND) INTO (INFIELD);
.
.
.
GO TO NEXT;
```

The first READ statement causes the first nondummy record in the data set whose key begins with “ABC” to be read into INFIELD; each time the second READ statement is executed, the nondummy record with the next higher key is retrieved. Repeated execution of the second READ statement could result in reading records from higher key classes, since no indication is given when the end of a key class is reached. It is your responsibility to check each key if you do not wish to read beyond the key class. Any subsequent execution of the first READ statement would reposition the file to the first record of the key class “ABC”.

If the data set contains no records with keys in the specified class, or if all the records with keys in the specified class are dummy records, the KEY condition is raised. The data set is then positioned either at the next record that has a higher key or at the end of the file.

The presence or absence of the GENKEY option affects the execution of a READ statement which supplies a source key that is shorter than the key length specified in the KEYLEN subparameter. This KEYLEN subparameter is found in the DD statement that defines the indexed data set. If you specify the GENKEY option, it causes the source key to be interpreted as a generic key, and the data set is positioned to the first nondummy record in the data set whose key begins with the source key. If you do not specify the GENKEY option, a READ statement's short source key is padded on the right with blanks to the specified key length, and the data set is positioned to the record that has this padded key (if such a record exists). For a WRITE statement, a short source key is always padded with blanks.

Use of the GENKEY option does not affect the result of supplying a source key whose length is greater than or equal to the specified key length. The source key, truncated on the right if necessary, identifies a specific record (whose key can be considered to be the only member of its class).

**NCP Option — Number of Channel Programs:** The NCP option specifies the number of incomplete input/output operations with the EVENT option that can be handled for the file at any one time.

►►NCP—(—n—)◄◄

For *n* you specify an integer in the range 1 through 99. If you do not specify anything, *n* defaults to 1.

For consecutive and regional sequential files, it is an error to allow more than the specified number of events to be outstanding.

For indexed files, any excess operations are queued, and no condition is raised. However, specifying the number of channel programs required can aid optimization of I/O with an indexed file. The NCP option has no effect with a regional direct file.

A file declared with ENVIRONMENT(VSAM) can never have more than one incomplete input/output operation at any one time. If you specify the NCP option for such a file, it is ignored. For information about the NCP option for VSAM with the ISAM compatibility interface, see "Using the VSAM Compatibility Interface" on page 225.

**TRKOFL Option — Track Overflow:** Track overflow is a feature of the operating system that can be incorporated at PL/I installation time; it requires the record overflow feature on the direct-access storage control unit. Track overflow allows a record to overflow from one track to another. It is useful in achieving a greater data-packing efficiency, and allows the size of a record to exceed the capacity of a track.

▶—TRKOFL—▶

Track overflow is not available for REGIONAL(3) or INDEXED data sets.

**COBOL Option — Data Interchange:** The COBOL option specifies that structures in the data set associated with the file will be mapped as they would be in a COBOL compiler. The COBOL structures can be synchronized or unsynchronized; it is your responsibility to ensure that the associated PL/I structure has the equivalent alignment stringency; that is, it must be ALIGNED or UNALIGNED, respectively.

▶—COBOL—▶

The following restrictions apply to the handling of a file with the COBOL option:

- You can only use a file with the COBOL option for READ INTO, WRITE FROM, and REWRITE FROM statements.
- You cannot pass the file name as an argument or assign it to a file variable.
- You must subscript any array variable to be transmitted.
- If a condition is raised during the execution of a READ statement, you cannot use the variable named in the INTO option in the ON-unit. If the completed INTO variable is required, there must be a normal return from the ON-unit.
- You can use the EVENT option only if the compiler determines that the PL/I and COBOL structure mappings are identical (that is, all elementary items have identical boundaries). If the mappings are not identical, or if the compiler cannot tell whether they are identical, an intermediate variable is created to represent the level-1 item as mapped by the COBOL algorithm. The PL/I

variable is assigned to the intermediate variable before a WRITE statement is executed, or assigned from it after a READ statement has been executed.

**SCALARVARYING Option — Varying-Length Strings:** You use the SCALARVARYING option in the input/output of varying-length strings; you can use it with records of any format.

►—SCALARVARYING—◄

When storage is allocated for a varying-length string, the compiler includes a 2-byte prefix that specifies the current length of the string. For an element varying-length string, this prefix is included on output, or recognized on input, only if SCALARVARYING is specified for the file.

When you use locate mode statements (LOCATE and READ SET) to create and read a data set with element varying-length strings, you must specify SCALARVARYING to indicate that a length prefix is present, since the pointer that locates the buffer is always assumed to point to the start of the length prefix.

When you specify SCALARVARYING and element varying-length strings are transmitted, you must allow two bytes in the record length to include the length prefix.

A data set created using SCALARVARYING should be accessed only by a file that also specifies SCALARVARYING.

You must not specify SCALARVARYING and CTLASA/CTL360 for the same file, as this causes the first data byte to be ambiguous.

**KEYLENGTH Option:** Use the KEYLENGTH option to specify the length of the recorded key for KEYED files where *n* is the length. You can specify KEYLENGTH for INDEXED or REGIONAL(3) files.

►—KEYLENGTH—(—*n*—)—◄

If you include the KEYLENGTH option in a VSAM file declaration for checking purposes, and the key length you specify in the option conflicts with the value defined for the data set, the UNDEFINEDFILE condition is raised.

## Data Set Types Used by PL/I Record I/O

Data sets with the RECORD attribute are processed by record-oriented data transmission in which data is transmitted to and from auxiliary storage exactly as it appears in the program variables; no data conversion takes place. A record in a data set corresponds to a variable in the program.

Table 16 on page 122 shows the facilities that are available with the various types of data sets that can be used with PL/I Record I/O.

The following chapters describe how to use Record I/O data sets for different types of data sets:

- Chapter 8, “Defining and Using Consecutive Data Sets” on page 129
- Chapter 9, “Defining and Using Indexed Data Sets” on page 163

- Chapter 10, “Defining and Using Regional Data Sets” on page 185
- Chapter 11, “Defining and Using VSAM Data Sets” on page 211
- Chapter 12, “Defining and Using Teleprocessing Data Sets” on page 255

Table 16. A Comparison of Data Set Types Available to PL/I Record I/O

	VSAM KSDS	VSAM ESDS	VSAM RRDS	INDEXED	CONSECUTIVE	REGIONAL (1)	REGIONAL (2)	REGIONAL (3)
SEQUENCE	Key order	Entry order	Num- bered	Key order	Entry order	By region	By region	By region
DEVICES	DASD	DASD	DASD	DASD	DASD, tape, card, etc.	DASD	DASD	DASD
ACCESS								
1 By key								
2 Sequential	123	123	123	12	2	12	12	12
3 Backward					3 tape only			
Alternate index access as above	123	123	No	No	No	No	No	No
How extended	With new keys	At end	In empty slots	With new keys	At end	In empty slots	With new keys	With new keys
SPANNED RECORDS	Yes	Yes	No	Yes	Yes	No	No	Yes
DELETION								
1 Space reusable	Yes, 1	No	Yes, 1	Yes, 2	No	Yes, 2	Yes, 2	Yes, 2
2 Space not reusable								

---

## Chapter 7. Using Libraries

Within the MVS operating system, the terms “partitioned data set” and “library” are synonymous and refer to a type of data set that can be used for the storage of other data sets (usually programs in the form of source, object or load modules). A library must be stored on direct-access storage and be wholly contained in one volume. It contains independent, consecutively organized data sets, called members. Each member has a unique name, not more than 8 characters long, which is stored in a directory that is part of the library. All the members of one library must have the same data characteristics because only one data set label is maintained.

You can create members individually until there is insufficient space left for a new entry in the directory, or until there is insufficient space for the member itself. You can access members individually by specifying the member name.

Use DD statements or their conversational mode equivalent to create and access members.

You can delete members by means of the IBM utility program IEHPROGM. This deletes the member name from the directory so that the member can no longer be accessed, but you cannot use the space occupied by the member itself again unless you recreate the library or compress the unused space using, for example, the IBM utility program IEBCOPY. If you attempt to delete a member by using the DISP parameter of a DD statement, it causes the whole data set to be deleted.

PL/I does not support VM MACLIBS as libraries.

### Types of libraries

You can use the following types of libraries with a PL/I program:

- The system program library SYS1.LINKLIB or its equivalent. This can contain all system processing programs such as compilers and the linkage editor.
- Private program libraries. These usually contain user-written programs. It is often convenient to create a temporary private library to store the load module output from the linkage editor until it is executed by a later job step in the same job. The temporary library will be deleted at the end of the job. Private libraries are also used for automatic library call by the linkage editor and the loader.
- The system procedure library SYS1.PROCLIB or its equivalent. This contains the job control procedures that have been cataloged for your installation.

### How to Use a Library

A PL/I program can use a library directly. If you are adding a new member to a library, its directory entry will be made by the operating system when the associated file is closed, using the member name specified as part of the data set name.

If you are accessing a member of a library, its directory entry can be found by the operating system from the member name that you specify as part of the data set name.

More than one member of the same library can be processed by the same PL/I program, but only one such output file can be open at any one time. You access different members by giving the member name in a DD statement.

---

## Creating a Library

To create a library include in your job step a DD statement containing the information given in Table 17. The information required is similar to that for a consecutively organized data set (see "Defining Files Using Record I/O" on page 150) except for the SPACE parameter.

*Table 17. Information Required When Creating a Library*

Information required	Parameter of DD statement
Type of device that will be used	UNIT=
Serial number of the volume that will contain the library	VOLUME=SER
Name of the library	DSNAME=
Amount of space required for the library	SPACE=
Disposition of the library	DISP=

## SPACE Parameter

The SPACE parameter in a DD statement that defines a library must always be of the form:

```
SPACE=(units,(quantity,increment,directory))
```

Although you can omit the third term (increment), indicating its absence by a comma, the last term, specifying the number of directory blocks to be allocated, must always be present.

The amount of auxiliary storage required for a library depends on the number and sizes of the members to be stored in it and on how often members will be added or replaced. (Space occupied by deleted members is not released.) The number of directory blocks required depends on the number of members and the number of aliases. You can specify an incremental quantity in the SPACE parameter that allows the operating system to obtain more space for the data set, if such is necessary at the time of creation or at the time a new member is added; the number of directory blocks, however, is fixed at the time of creation and cannot be increased.

For example, the DD statement:

```
//PDS DD UNIT=SYSDA,VOL=SER=3412,  
// DSNAME=ALIB,  
// SPACE=(CYL,(5,,10)),  
// DISP=(,CATLG)
```

requests the job scheduler to allocate 5 cylinders of the DASD with a volume serial number 3412 for a new library name ALIB, and to enter this name in the system catalog. The last term of the SPACE parameter requests that part of the space allocated to the data set be reserved for ten directory blocks.

---

## Creating and Updating a Library Member

The members of a library must have identical characteristics. Otherwise, you might later have difficulty retrieving them. Identical characteristics are necessary because the volume table of contents (VTOC) will contain only one data set control block (DSCB) for the library and not one for each member. When using a PL/I program to create a member, the operating system creates the directory entry; you cannot place information in the user data field.

When creating a library and a member at the same time, your DD statement must include all the parameters listed under “Creating a Library” on page 124 (although you can omit the DISP parameter if the data set is to be temporary). The DSNAMES parameter must include the member name in parentheses. For example, DSNAMES=ALIB(MEM1) names the member MEM1 in the data set ALIB. If the member is placed in the library by the linkage editor, you can use the linkage editor NAME statement or the NAME compile-time option instead of including the member name in the DSNAMES parameter. You must also describe the characteristics of the member (record format, etc.) either in the DCB parameter or in your PL/I program. These characteristics will also apply to other members added to the data set.

When creating a member to be added to an existing library, you do not need the SPACE parameter. The original space allocation applies to the whole of the library and not to an individual member. Furthermore, you do not need to describe the characteristics of the member, since these are already recorded in the DSCB for the library.

To add two more members to a library in one job step, you must include a DD statement for each member, and you must close one file that refers to the library before you open another.

## Examples

The use of the cataloged procedure IEL1C to compile a simple PL/I program and place the object module in a new library named EXLIB is shown in Figure 23 on page 126. The DD statement that defines the new library and names the object module overrides the DD statement SYSLIN in the cataloged procedure. (The PL/I program is a function procedure that, given two values in the form of the character string produced by the TIME built-in function, returns the difference in milliseconds.)

The use of the cataloged procedure IEL1CL to compile and link-edit a PL/I program and place the load module in the existing library HPU8.CCLM is shown in Figure 24 on page 126.

---

```

//OPT10#1 JOB
//TR      EXEC  IEL1C
//PLI.SYSLIN DD UNIT=SYSDA,DSNAME=HPU8.EXLIB(ELAPSE),
//      SPACE=(TRK,(1,,1)),DISP=(NEW,CATLG)
//PLI.SYSLIN DD *
      ELAPSE: PROC(TIME1,TIME2);
            DCL (TIME1,TIME2) CHAR(9),
            H1 PIC '99' DEF TIME1,
            M1 PIC '99' DEF TIME1 POS(3),
            MS1 PIC '99999' DEF TIME1 POS(5),
            H2 PIC '99' DEF TIME2,
            M2 PIC '99' DEF TIME2 POS(3),
            MS2 PIC '99999' DEF TIME2 POS(5),
            ETIME FIXED DEC(7);
            IF H2<H1 THEN H2=H2+24;
            ETIME=((H2*60+M2)*60000+MS2)-((H1*60+M1)*60000+MS1);
            RETURN(ETIME);
      END ELAPSE;
/*

```

---

*Figure 23. Creating New Libraries for Compiled Object Modules*

---

```

//OPT10#2 JOB
//TRLE    EXEC  IEL1CL
//PLI.SYSLIN DD *
      MNAME: PROC  OPTIONS(MAIN);
            .
            .
            .
            program
            .
            .
            .
      END MNAME;
/*
//LKED.SYSLMOD DD  DSNAME=HPU8.CCLM(DIRLIST),DISP=OLD

```

---

*Figure 24. Placing a Load Module in an Existing Library*

To use a PL/I program to add or delete one or more records within a member of a library, you must rewrite the entire member in another part of the library. This is rarely an economic proposition, since the space originally occupied by the member cannot be used again. You must use two files in your PL/I program, but both can be associated with the same DD statement. The program shown in Figure 26 on page 127 updates the member created by the program in Figure 25 on page 127. It copies all the records of the original member except those that contain only blanks.

---

```

//OPT10#3 JOB
//TREX EXEC IELICLG
//PLI.SYSIN DD *
  NMEM: PROC OPTIONS(MAIN);
    DCL IN FILE RECORD SEQUENTIAL INPUT,
        OUT FILE RECORD SEQUENTIAL OUTPUT,
        P POINTER,
        IOFIELD CHAR(80) BASED(P),
        EOF BIT(1) INIT('0'B);
    OPEN FILE(IN),FILE (OUT);
    ON ENDFILE(IN) EOF='1'B;
    READ FILE(IN) SET(P);
    DO WHILE (~EOF);
    PUT FILE(SYSPRINT) SKIP EDIT (IOFIELD) (A);
    WRITE FILE(OUT) FROM(IOFIELD);
    READ FILE(IN) SET(P);
    END;
    CLOSE FILE(IN),FILE(OUT);
  END NMEM;
/*
//GO.OUT DD UNIT=SYSDA,DSNAME=HPU8.ALIB(NMEM),
//  DISP=(NEW,CATLG),SPACE=(TRK,(1,1,1)),
//  DCB=(RECFM=FB,BLKSIZE=3600,LRECL=80)
//GO.IN DD *
  MEM: PROC OPTIONS(MAIN);
    /* this is an incomplete dummy library member */

```

---

*Figure 25. Creating a Library Member in a PL/I Program*

---

```

//OPT10#4 JOB
//TREX EXEC IELICLG
//PLI.SYSIN DD *
  UPDTM: PROC OPTIONS(MAIN);
    DCL (OLD,NEW) FILE RECORD SEQUENTIAL,
        EOF BIT(1) INIT('0'B),
        DATA CHAR(80);
    ON ENDFILE(OLD) EOF = '1'B;
    OPEN FILE(OLD) INPUT,FILE(NEW) OUTPUT TITLE('OLD');
    READ FILE(OLD) INTO(DATA);
    DO WHILE (~EOF);
    PUT FILE(SYSPRINT) SKIP EDIT (DATA) (A);
    IF DATA= ' ' THEN ;
    ELSE WRITE FILE(NEW) FROM(DATA);
    READ FILE(OLD) INTO(DATA);
    END;
    CLOSE FILE(OLD),FILE(NEW);
  END UPDTM;
/*
//GO.OLD DD DSNAME=HPU8.ALIB(NMEM),DISP=(OLD,KEEP)

```

---

*Figure 26. Updating a Library Member*

---

## Extracting Information from a Library Directory

The directory of a library is a series of records (entries) at the beginning of the data set. There is at least one directory entry for each member. Each entry contains a member name, the relative address of the member within the library, and a variable amount of user data.

User data is information inserted by the program that created the member. An entry that refers to a member (load module) written by the linkage editor includes user data in a standard format, described in the systems manuals.

If you use a PL/I program to create a member, the operating system creates the directory entry for you and you cannot write any user data. However, you can use assembler language macro instructions to create a member and write your own user data. The method for using macro instructions to do this is described in the data management manuals.

---

## Chapter 8. Defining and Using Consecutive Data Sets

This chapter covers consecutive data set organization and the ENVIRONMENT options that define consecutive data sets for stream and record-oriented data transmission. It then covers how to create, access, and update consecutive data sets for each type of transmission.

In a data set with consecutive organization, records are organized solely on the basis of their successive physical positions; when the data set is created, records are written consecutively in the order in which they are presented. You can retrieve the records only in the order in which they were written, or, for RECORD I/O only, also in the reverse order when using the BACKWARDS attribute. See Table 15 on page 111 for valid file attributes and ENVIRONMENT options for consecutive data sets.

VM supports consecutive data set organization, and you can use PL/I to access these types of files. The examples in this chapter are given using JCL. However, the information presented in the JCL examples is applicable to the FILEDEF VM command you issue. For more information on the FILEDEF command, see the *VM/ESA CMS Command Reference* and the *VM/ESA CMS User's Guide*.

---

### Using Stream-Oriented Data Transmission

This section covers how to define data sets for use with PL/I files that have the STREAM attribute. It covers the ENVIRONMENT options you can use and how to create and access data sets. The essential parameters of the DD statements you use in creating and accessing these data sets are summarized in tables, and several examples of PL/I programs are included to illustrate the text.

Data sets with the STREAM attribute are processed by stream-oriented data transmission, which allows your PL/I program to ignore block and record boundaries and treat a data set as a continuous stream of data values in character or graphic form.

You create and access data sets for stream-oriented data transmission using the list-, data-, and edit-directed input and output statements described in the *PL/I for MVS & VM Language Reference*.

For output, PL/I converts the data items from program variables into character form if necessary, and builds the stream of characters or graphics into records for transmission to the data set.

For input, PL/I takes records from the data set and separates them into the data items requested by your program, converting them into the appropriate form for assignment to program variables.

You can use stream-oriented data transmission to read or write graphic data. There are terminals, printers, and data-entry devices that, with the appropriate programming support, can display, print, and enter graphics. You must be sure that your data is in a format acceptable for the intended device, or for a print utility program.

## Defining Files Using Stream I/O

You define files for stream-oriented data transmission by a file declaration with the following attributes:

```
DCL filename FILE STREAM
      INPUT | {OUTPUT [PRINT]}
      ENVIRONMENT(options);
```

Default file attributes are shown in Table 15 on page 111; the FILE attribute is described in the *PL/I for MVS & VM Language Reference*. The PRINT attribute is described further in “Using PRINT Files with Stream I/O” on page 138. Options of the ENVIRONMENT attribute are discussed below.

## Specifying ENVIRONMENT Options

Table 15 on page 111 summarizes the ENVIRONMENT options. The options applicable to stream-oriented data transmission are:

```
CONSECUTIVE
F|FB|FS|FBS|V|VB|D|DB|U
RECSIZE(record-length)
BLKSIZE(block-size)
BUFFERS(n)
GRAPHIC
LEAVE
REREAD
ASCII
BUFOFF[(n)]
```

BLKSIZE and BUFFERS are described in Chapter 6, “Using Data Sets and Files,” beginning on page 115. LEAVE, REREAD, ASCII, and BUFOFF are described later in this chapter, beginning on page 153. Descriptions of the rest of these options follow immediately below.

### CONSECUTIVE

STREAM files must have CONSECUTIVE data set organization; however, it is not necessary to specify this in the ENVIRONMENT options since CONSECUTIVE is the default data set organization. The CONSECUTIVE option for STREAM files is the same as that described in “Data Set Organization” on page 106.

►►—CONSECUTIVE—◄◄

### Record format options

Although record boundaries are ignored in stream-oriented data transmission, record format is important when creating a data set. This is not only because record format affects the amount of storage space occupied by the data set and the efficiency of the program that processes the data, but also because the data set can later be processed by record-oriented data transmission.

Having specified the record format, you need not concern yourself with records and blocks as long as you use stream-oriented data transmission. You can consider your data set a series of characters or graphics arranged in lines, and you can use the SKIP option or format item (and, for a PRINT file, the PAGE and LINE options and format items) to select a new line.



Records can have one of the following formats, which are described in “Record Formats” on page 103.

Fixed-length	F	unblocked
	FB	blocked
	FBS	blocked, standard
	FS	unblocked, standard
Variable-length	V	unblocked
	VB	blocked
	D	unblocked ASCII
	DB	blocked ASCII
Undefined-length	U	(cannot be blocked)

Blocking and deblocking of records are performed automatically.

## RECSIZE

RECSIZE for stream-oriented data transmission is the same as that described in “Specifying Characteristics in the ENVIRONMENT Attribute” on page 110. Additionally, a value specified by the LINESIZE option of the OPEN statement overrides a value specified in the RECSIZE option. LINESIZE is discussed in the *PL/I for MVS & VM Language Reference*.

Additional record-size considerations for list- and data-directed transmission of graphics are given in the *PL/I for MVS & VM Language Reference*.

## Defaults for Record Format, BLKSIZE, and RECSIZE

If you do not specify the record format, BLKSIZE, or RECSIZE option in the ENVIRONMENT attribute, or in the associated DD statement or data set label, the following action is taken:

### Input files:

Defaults are applied as for record-oriented data transmission, described in “Record Format, BLKSIZE, and RECSIZE Defaults” on page 117.

### Output files:

#### Record format:

Set to VB-format, or if ASCII option specified, to DB-format.

**Record length:**

The specified or default LINESIZE value is used:

**PRINT files:**

F, FB, FBS, or U: line size + 1  
 V, VB, D, or DB: line size + 5

**Non-PRINT files:**

F, FB, FBS, or U: linesize  
 V, VB, D, or DB: linesize + 4

**Block size:**

F, FB, or FBS: record length  
 V or VB: record length + 4  
 D or DB: record length + block prefix  
 (see "Information Interchange Codes" on page 103)

**GRAPHIC Option**

You must specify the GRAPHIC option of the ENVIRONMENT attribute if you use DBCS variables or DBCS constants in GET and PUT statements for list- and data-directed I/O. You can also specify the GRAPHIC option for edit-directed I/O.

▶—GRAPHIC—▶

The ERROR condition is raised for list- and data-directed I/O if you have graphics in input or output data and do not specify the GRAPHIC option.

For edit-directed I/O, the GRAPHIC option specifies that left and right delimiters are added to DBCS variables and constants on output, and that input graphics will have left and right delimiters. If you do not specify the GRAPHIC option, left and right delimiters are not added to output data, and input graphics do not require left and right delimiters. When you do specify the GRAPHIC option, the ERROR condition is raised if left and right delimiters are missing from the input data.

For information on the graphic data type, and on the G-format item for edit-directed I/O, see the *PL/I for MVS & VM Language Reference*.

**Creating a Data Set with Stream I/O**

To create a data set, you must give the operating system certain information either in your PL/I program or in the DD statement that defines the data set. The following paragraphs indicate the essential information, and discuss some of the optional information you can supply.

**Essential Information**

You must supply the following information, summarized in Table 18 on page 133, when creating a data set:

- Device that will write your data set (UNIT, SYSOUT, or VOLUME parameter of DD statement).
- Block size: You can specify the block size either in your PL/I program (ENVIRONMENT attribute or LINESIZE option of the OPEN statement) or in the DD statement (BLKSIZE subparameter). If you do not specify a record length,

unblocked records are the default and the record length is determined from the block size. If you do not specify a record format, U-format is the default (except for PRINT files when V-format is the default; see “Controlling Printed Line Length” on page 139).

If you want to keep a magnetic-tape or direct-access data set (that is, you do not want the operating system to delete it at the end of your job), the DD statement must name the data set and indicate how it is to be disposed of (DSNAME and DISP parameters). The DISP parameter alone will suffice if you want to use the data set in a later step but will not need the data set after the end of your job.

*Table 18. Creating a data set with stream I/O: essential parameters of the DD statement*

Storage device	When required	What you must state	Parameters
All	Always	Output device	UNIT= or SYSOUT= or VOLUME=REF=
		Block size <sup>1</sup>	DCB=(BLKSIZE=...)
Direct access only	Always	Storage space required	SPACE=
Magnetic tape only	Data set not first in volume and for magnetic tapes that do not have standard labels	Sequence number	LABEL=
Direct access and standard labeled magnetic tape	Data set to be used by another job step but not required at end of job	Disposition	DISP=
	Data set to be kept after end of job	Disposition	DISP=
	Data set to be on particular volume	Name of data set	DSNAME=
		Volume serial number	VOLUME=SER or VOLUME=REF=

<sup>1</sup>Alternatively, you can specify the block size in your PL/I program by using either the ENVIRONMENT attribute or the LINESIZE option.

When creating a data set on a direct-access device, you must specify the amount of space required for it (SPACE parameter of DD statement).

If you want your data set stored on a particular magnetic-tape or direct-access device, you must indicate the volume serial number in the DD statement (SER or REF subparameter of VOLUME parameter). If you do not supply a serial number for a magnetic-tape data set that you want to keep, the operating system will allocate one, inform the operator, and print the number on your program listing.

If your data set is to follow another data set on a magnetic-tape volume, you must use the LABEL parameter of the DD statement to indicate its sequence number on the tape.

## Examples

The use of edit-directed stream-oriented data transmission to create a data set on a direct access storage device is shown in Figure 27 on page 134. The data read from the input stream by the file SYSIN includes a field VREC that contains five unnamed 7-character subfields; the field NUM defines the number of these subfields that contain information. The output file WORK transmits to the data set

the whole of the field FREC and only those subfields of VREC that contain information.

---

```
//EX7#2 JOB
//STEP1 EXEC IEL1CLG
//PLI.SYSIN DD *
  PEOPLE: PROC OPTIONS(MAIN);
    DCL WORK FILE STREAM OUTPUT,
      1 REC,
      2 FREC,
      3 NAME CHAR(19),
      3 NUM CHAR(1),
      3 PAD CHAR(25),
      2 VREC CHAR(35),
      EOF BIT(1) INIT('0'B),
      IN CHAR(80) DEF REC;
    ON ENDFILE(SYSIN) EOF='1'B;
    OPEN FILE(WORK) LINESIZE(400);
    GET FILE(SYSIN) EDIT(IN)(A(80));
    DO WHILE (-EOF);
      PUT FILE(WORK) EDIT(IN)(A(45+7*NUM));
      GET FILE(SYSIN) EDIT(IN)(A(80));
    END;
    CLOSE FILE(WORK);
  END PEOPLE;

/*
//GO.WORK DD DSN=HPU8.PEOPLE,DISP=(NEW,CATLG),UNIT=SYSDA,
//          SPACE=(TRK,(1,1))
//GO.SYSIN DD *
R.C.ANDERSON      0 202848 DOCTOR
B.F.BENNETT      2 771239 PLUMBER          VICTOR HAZEL
R.E.COLE         5 698635 COOK            ELLEN VICTOR JOAN ANN OTTO
J.F.COOPER       5 418915 LAWYER          FRANK CAROL DONALD NORMAN BRENDA
A.J.CORNELL      3 237837 BARBER          ALBERT ERIC JANET
E.F.FERRIS       4 158636 CARPENTER       GERALD ANNA MARY HAROLD
/*
```

---

Figure 27. Creating a Data Set with Stream-Oriented Data Transmission

Figure 28 on page 135 shows an example of a program using list-directed output to write graphics to a stream file. It assumes that you have an output device that can print graphic data. The program reads employee records and selects persons living in a certain area. It then edits the address field, inserting one graphic blank between each address item, and prints the employee number, name, and address.

```

//EX7#3 JOB
//STEP1 EXEC IEL1CLG
//PLI.SYSIN DD *
% PROCESS GRAPHIC;
  XAMPLE1: PROC OPTIONS(MAIN);
    DCL INFILE FILE INPUT RECORD,
        OUTFILE FILE OUTPUT STREAM ENV(GRAPHIC);
/* GRAPHIC OPTION MEANS DELIMITERS WILL BE INSERTED ON OUTPUT FILES. */
    DCL
      1 IN,
        3 EMPNO CHAR(6),
        3 SHIFT1 CHAR(1),
        3 NAME,
          5 LAST G(7),
          5 FIRST G(7),
        3 SHIFT2 CHAR(1),
        3 ADDRESS,
          5 ZIP CHAR(6),
          5 SHIFT3 CHAR(1),
          5 DISTRICT G(5),
          5 CITY G(5),
          5 OTHER G(8),
          5 SHIFT4 CHAR(1);
    DCL EOF BIT(1) INIT('0'B);
    DCL ADDRWK G(20);
    ON ENDFILE (INFILE) EOF = '1'B;
    READ FILE(INFILE) INTO(IN);
    DO WHILE(~EOF);
      DO;
        IF SUBSTR(ZIP,1,3)~='300'
          THEN LEAVE;
        L=0;
        ADDRWK=DISTRICT;
        DO I=1 TO 5;
          IF SUBSTR(DISTRICT,I,1)= <' ' G>
            THEN LEAVE; /* SUBSTR BIF PICKS UP */
          END; /* THE ITH GRAPHIC CHAR */
          L=L+I; /* IN DISTRICT */
          SUBSTR(ADDRWK,L,5)=CITY;
          DO I=1 TO 5;
            IF SUBSTR(CITY,I,1)= <' ' G>
              THEN LEAVE;
            END;
            L=L+I;
            SUBSTR(ADDRWK,L,8)=OTHER;
            PUT FILE(OUTFILE) SKIP /* THIS DATA SET */
            EDIT(EMPNO,IN.LAST,FIRST,ADDRWK) /* REQUIRES UTILITY */
            (A(8),G(7),G(7),X(4),G(20)); /* TO PRINT GRAPHIC */
            /* DATA */
          END; /* END OF NON-ITERATIVE DO */
        READ FILE(INFILE) INTO (IN);
        END; /* END OF DO WHILE(~EOF) */
      END XAMPLE1;
/*
//GO.OUTFILE DD SYSOUT=A,DCB=(RECFM=VB,LRECL=121,BLKSIZE=129)
//GO.INFILE DD *
ABCDEF<山山山山山山日日日日日>300099< 3 3 3 3 3 3 >
ABCD <山山山山 日日日日 >300011< 3 3 3 3 >
/*

```

Figure 28. Writing Graphic Data to a Stream File

## Accessing a Data Set with Stream I/O

A data set accessed using stream-oriented data transmission need not have been created by stream-oriented data transmission, but it must have CONSECUTIVE organization, and all the data in it must be in character or graphic form. You can open the associated file for input, and read the records the data set contains; or you can open the file for output, and extend the data set by adding records at the end.

To access a data set, you must identify it to the operating system in a DD statement. Table 19 summarizes the DD statement parameters needed to access a consecutive data set.

*Table 19. Accessing a Data Set with Stream I/O: Essential Parameters of the DD Statement*

When required	What you must state	Parameters
Always	Name of data set	DSNAME=
	Disposition of data set	DISP=
If data set not cataloged (all devices)	Input device	UNIT= or VOLUME=REF=
If data set not cataloged (standard labeled magnetic tape and direct access)	Volume serial number	VOLUME=SER=
Magnetic tape (if data set not first in volume or which does not have standard labels)	Sequence number	LABEL=
If data set does not have standard labels	Block size <sup>1</sup>	DCB=(BLKSIZE=.

<sup>1</sup>Or you could specify the block size in your PL/I program by using either the ENVIRONMENT attribute or the LINESIZE option.

The following paragraphs describe the essential information you must include in the DD statement, and discuss some of the optional information you can supply. The discussions do not apply to data sets in the input stream.

### Essential Information

If the data set is cataloged, you need supply only the following information in the DD statement:

- The name of the data set (DSNAME parameter). The operating system locates the information describing the data set in the system catalog, and, if necessary, requests the operator to mount the volume containing it.
- Confirmation that the data set exists (DISP parameter). If you open the data set for output with the intention of extending it by adding records at the end, code DISP=MOD; otherwise, opening the data set for output results in it being overwritten.

If the data set is not cataloged, you must, in addition, specify the device that will read the data set and, for magnetic-tape and direct-access devices, give the serial number of the volume that contains the data set (UNIT and VOLUME parameters).

If the data set follows another data set on a magnetic-tape volume, you must use the LABEL parameter of the DD statement to indicate its sequence number on the tape.

***Magnetic Tape without IBM Standard Labels:*** If a magnetic-tape data set has nonstandard labels or is unlabeled, you must specify the block size either in your PL/I program (ENVIRONMENT attribute) or in the DD statement (BLKSIZE subparameter). The DSNAME parameter is not essential if the data set is not cataloged.

PL/I includes no facilities for processing nonstandard labels, which appear to the operating system as data sets preceding or following your data set. You can either process the labels as independent data sets or use the LABEL parameter of the DD statement to bypass them. To bypass the labels, code LABEL=(2,NL) or LABEL=(,BLP)

### **Record Format**

When using stream-oriented data transmission to access a data set, you do not need to know the record format of the data set (except when you must specify a block size); each GET statement transfers a discrete number of characters or graphics to your program from the data stream.

If you do give record-format information, it must be compatible with the actual structure of the data set. For example, if a data set is created with F-format records, a record size of 600 bytes, and a block size of 3600 bytes, you can access the records as if they are U-format with a maximum block size of 3600 bytes; but if you specify a block size of 3500 bytes, your data will be truncated.

### **Example**

The program in Figure 29 on page 138 reads the data set created by the program in Figure 27 on page 134 and uses the file SYSPRINT to list the data it contains. (For details on SYSPRINT, see “Using SYSIN and SYSPRINT Files” on page 142.) Each set of data is read, by the GET statement, into two variables: FREC, which always contains 45 characters; and VREC, which always contains 35 characters. At each execution of the GET statement, VREC consists of the number of characters generated by the expression 7\*NUM, together with sufficient blanks to bring the total number of characters to 35. The DISP parameter of the DD statement could read simply DISP=OLD; if DELETE is omitted, an existing data set will not be deleted.

---

```

//EX7#5 JOB
//STEP1 EXEC IEL1CLG
//PLI.SYSIN DD *
  PEOPLE: PROC OPTIONS(MAIN);
    DCL WORK FILE STREAM INPUT,
      1 REC,
      2 FREC,
      3 NAME CHAR(19),
      3 NUM CHAR(1),
      3 SERNO CHAR(7),
      3 PROF CHAR(18),
      2 VREC CHAR(35),
    IN CHAR(80) DEF REC,
    EOF BIT(1) INIT('0'B);
  ON ENDFILE(WORK) EOF='1'B;
  OPEN FILE(WORK);
  GET FILE(WORK) EDIT(IN,VREC)(A(45),A(7*NUM));
  DO WHILE (~EOF);
  PUT FILE(SYSPRINT) SKIP EDIT(IN)(A);
  GET FILE(WORK) EDIT(IN,VREC)(A(45),A(7*NUM));
  END;
  CLOSE FILE(WORK);
  END PEOPLE;

/*
//GO.WORK DD DSN=HPU8.PEOPLE,DISP=(OLD,DELETE)

```

---

Figure 29. Accessing a Data Set with Stream-Oriented Data Transmission

## Using PRINT Files with Stream I/O

Both the operating system and the PL/I language include features that facilitate the formatting of printed output. The operating system allows you to use the first byte of each record for a print control character. The control characters, which are not printed, cause the printer to skip to a new line or page. (Tables of print control characters are given in Figure 33 on page 153 and Figure 34 on page 153.)

In a PL/I program, the use of a PRINT file provides a convenient means of controlling the layout of printed output from stream-oriented data transmission. The compiler automatically inserts print control characters in response to the PAGE, SKIP, and LINE options and format items.

You can apply the PRINT attribute to any STREAM OUTPUT file, even if you do not intend to print the associated data set directly. When a PRINT file is associated with a magnetic-tape or direct-access data set, the print control characters have no effect on the layout of the data set, but appear as part of the data in the records.

The compiler reserves the first byte of each record transmitted by a PRINT file for an American National Standard print control character, and inserts the appropriate characters automatically.

A PRINT file uses only the following five print control characters:

**Character Action**

	Space 1 line before printing (blank character)
0	Space 2 lines before printing
-	Space 3 lines before printing
+	No space before printing
1	Start new page

The compiler handles the PAGE, SKIP, and LINE options or format items by padding the remainder of the current record with blanks and inserting the appropriate control character in the next record. If SKIP or LINE specifies more than a 3-line space, the compiler inserts sufficient blank records with appropriate control characters to accomplish the required spacing. In the absence of a print control option or format item, when a record is full the compiler inserts a blank character (single line space) in the first byte of the next record.

If a PRINT file is being transmitted to a terminal, the PAGE, SKIP, and LINE options will never cause more than 3 lines to be skipped, unless formatted output is specified. (For information about TSO see "Using the PL/I Command" on page 59, and for information about VM see "PLIOPT Command Options" on page 75.)

**Controlling Printed Line Length**

You can limit the length of the printed line produced by a PRINT file either by specifying a record length in your PL/I program (ENVIRONMENT attribute) or in a DD statement, or by giving a line size in an OPEN statement (LINESIZE option). The record length must include the extra byte for the print control character, that is, it must be 1 byte larger than the length of the printed line (5 bytes larger for V-format records). The value you specify in the LINESIZE option refers to the number of characters in the printed line; the compiler adds the print control character.

The blocking of records has no effect on the appearance of the output produced by a PRINT file, but it does result in more efficient use of auxiliary storage when the file is associated with a data set on a magnetic-tape or direct-access device. If you use the LINESIZE option, ensure that your line size is compatible with your block size. For F-format records, block size must be an exact multiple of (line size+1); for V-format records, block size must be at least 9 bytes greater than line size.

Although you can vary the line size for a PRINT file during execution by closing the file and opening it again with a new line size, you must do so with caution if you are using the PRINT file to create a data set on a magnetic-tape or direct-access device. You cannot change the record format that is established for the data set when the file is first opened. If the line size you specify in an OPEN statement conflicts with the record format already established, the UNDEFINEDFILE condition is raised. To prevent this, either specify V-format records with a block size at least 9 bytes greater than the maximum line size you intend to use, or ensure that the first OPEN statement specifies the maximum line size. (Output destined for the printer can be stored temporarily on a direct-access device, unless you specify a printer by using UNIT=, even if you intend it to be fed directly to the printer.)

Since PRINT files have a default line size of 120 characters, you need not give any record format information for them. In the absence of other information, the compiler assumes V-format records. The complete default information is:

```
BLKSIZE=129
LRECL=125
RECFM=VBA.
```

**Example:** Figure 30 on page 141 illustrates the use of a PRINT file and the printing options of stream-oriented data transmission statements to format a table and write it onto a direct-access device for printing on a later occasion. The table comprises the natural sines of the angles from 0° to 359° 54' in steps of 6'.

The statements in the ENDPAGE ON-unit insert a page number at the bottom of each page, and set up the headings for the following page.

The DD statement defining the data set created by this program includes no record-format information. The compiler infers the following from the file declaration and the line size specified in the statement that opens the file TABLE:

```
Record format =  V
                (the default for a PRINT file).
Record size   =  98
                (line size + 1 byte for print control character + 4 bytes for
                record control field).
Block size    =  102
                (record length + 4 bytes for block control field).
```

The program in Figure 36 on page 162 uses record-oriented data transmission to print the table created by the program in Figure 30 on page 141.

### Overriding the Tab Control Table

Data-directed and list-directed output to a PRINT file are aligned on preset tabulator positions. See Figure 18 on page 90 and Figure 31 on page 142 for examples of declaring a tab table. The definitions of the fields in the table are as follows:

**OFFSET OF TAB COUNT:**

Halfword binary integer that gives the offset of "Tab count," the field that indicates the number of tabs to be used.

**PAGESIZE:**

Halfword binary integer that defines the default page size. This page size is used for dump output to the PLIDUMP data set as well as for stream output.

**LINESIZE:** Halfword binary integer that defines the default line size.

**PAGELength:**

Halfword binary integer that defines the default page length for printing at a terminal. For TSO and VM, the value 0 indicates unformatted output.

**FILLERS:** Three halfword binary integers; reserved for future use.

**TAB COUNT:**

Halfword binary integer that defines the number of tab position entries in the table (maximum 255). If tab count = 0, any specified tab positions are ignored.

---

```

%PROCESS INT F(I) AG A(F) ESD MAP OP STG NEST X(F) SOURCE ;
%PROCESS LIST;

SINE: PROC OPTIONS(MAIN);
DCL TABLE      FILE STREAM OUTPUT PRINT;
DCL DEG         FIXED DEC(5,1) INIT(0); /* INIT(0) FOR ENDPAGE */
DCL MIN         FIXED DEC(3,1);
DCL PGNO       FIXED DEC(2)  INIT(0);
DCL ONCODE     BUILTIN;

ON ERROR
  BEGIN;
  ON ERROR SYSTEM;
  DISPLAY ('ONCODE = ' || ONCODE);
END;

ON ENDPAGE(TABLE)
  BEGIN;
  DCL I;
  IF PGNO ^= 0 THEN
    PUT FILE(TABLE) EDIT ('PAGE',PGNO)
      (LINE(55),COL(80),A,F(3));
  IF DEG ^= 360 THEN
    DO;
    PUT FILE(TABLE) PAGE EDIT ('NATURAL SINES') (A);
    IF PGNO ^= 0 THEN
      PUT FILE(TABLE) EDIT ((I DO I = 0 TO 54 BY 6)
        (SKIP(3),10 F(9)));

    PGNO = PGNO + 1;
    END;
  ELSE
    PUT FILE(TABLE) PAGE;
  END;

OPEN FILE(TABLE) PAGESIZE(52) LINESIZE(93);
SIGNAL ENDPAGE(TABLE);

PUT FILE(TABLE) EDIT
  ((DEG,(SIND(DEG+MIN) DO MIN = 0 TO .9 BY .1) DO DEG = 0 TO 359))
  (SKIP(2), 5 (COL(1), F(3), 10 F(9,4) ));
PUT FILE(TABLE) SKIP(52);
END SINE;

```

---

*Figure 30. Creating a Print File Via Stream Data Transmission. The example in Figure 36 on page 162 will print the resultant file.*

#### Tab1–Tabn:

*n* halfword binary integers that define the tab positions within the print line. The first position is numbered 1, and the highest position is numbered 255. The value of each tab should be greater than that of the tab preceding it in the table; otherwise, it is ignored. The first data field in the printed output begins at the next available tab position.

You can override the default PL/I tab settings for your program by causing the linkage editor to resolve an external reference to PLITABS. To cause the reference to be resolved, supply a table with the name PLITABS, in the format described above.

There are two methods of supplying the tab table. One method is to include a PL/I structure in your source program with the name PLITABS, which you must declare to be STATIC EXTERNAL. An example of the PL/I structure is shown in Figure 31 on page 142. This example creates three tab settings, in positions 30, 60, and 90, and uses the defaults for page size and line size. Note that TAB1 identifies the position of the second item printed on a line; the first item on a line always starts at

the left margin. The first item in the structure is the offset to the NO\_OF\_TABS field; FILL1, FILL2, and FILL3 can be omitted by adjusting the offset value by -6.

The second method is to create an assembler language control section named PLITABS, equivalent to the structure shown in Figure 31, and to include it when link-editing your PL/I program.

---

```
DCL 1 PLITABS STATIC EXT,  
  2 (OFFSET INIT(14),  
    PAGESIZE INIT(60),  
    LINESIZE INIT(120),  
    PAGELENGTH INIT(0),  
    FILL1 INIT(0),  
    FILL2 INIT(0),  
    FILL3 INIT(0),  
    NO_OF_TABS INIT(3),  
    TAB1 INIT(30),  
    TAB2 INIT(60),  
    TAB3 INIT(90)) FIXED BIN(15,0);
```

---

Figure 31. PL/I Structure PLITABS for Modifying the Preset Tab Settings

## Using SYSIN and SYSPRINT Files

If you code a GET statement without the FILE option in your program, the compiler inserts the file name SYSIN. If you code a PUT statement without the FILE option, the compiler inserts the name SYSPRINT.

If you do not declare SYSPRINT, the compiler gives the file the attribute PRINT in addition to the normal default attributes; the complete set of attributes will be:

```
FILE STREAM OUTPUT PRINT EXTERNAL
```

Since SYSPRINT is a PRINT file, the compiler also supplies a default line size of 120 characters and a V-format record. You need give only a minimum of information in the corresponding DD statement; if your installation uses the usual convention that the system output device of class A is a printer, the following is sufficient:

```
//SYSPRINT DD SYSOUT=A
```

**Note:** SYSIN and SYSPRINT are established in the User Exit during initialization. IBM-supplied defaults for SYSIN and SYSPRINT are directed to the terminal.

You can override the attributes given to SYSPRINT by the compiler by explicitly declaring or opening the file. For more information about the interaction between SYSPRINT and the Language Environment for MVS & VM message file option, see the *Language Environment for MVS & VM Programming Guide*.

The compiler does not supply any special attributes for the input file SYSIN; if you do not declare it, it receives only the default attributes. The data set associated with SYSIN is usually in the input stream; if it is not in the input stream, you must supply full DD information.

For more information about SYSPRINT, see "SYSPRINT Considerations" on page 92.

---

## Controlling Input from the Terminal

You can enter data at the terminal for an input file in your PL/I program if you:

1. Declare the input file explicitly or implicitly with the CONSECUTIVE environment option (all stream files meet this condition), *and*
2. Allocate the input file to the terminal.

You can usually use the standard default input file SYSIN because it is a stream file and can be allocated to the terminal. In TSO, you can allocate SYSIN to the terminal in your logon procedure. In VM, SYSIN is allocated to the terminal by the IBM-supplied User Exit.

You are prompted for input to stream files by a colon (:). You will see the colon each time a GET statement is executed in the program. The GET statement causes the system to go to the next line. You can then enter the required data. If you enter a line that does not contain enough data to complete execution of the GET statement, a further prompt, which is a plus sign followed by a colon (+:), is displayed.

By adding a hyphen to the end of any line that is to continue, you can delay transmission of the data to your program until you enter two or more lines. The hyphen is an explicit continuation character in TSO.

If you include output statements that prompt you for input in your program, you can inhibit the initial system prompt by ending your own prompt with a colon. For example, the GET statement could be preceded by a PUT statement such as:

```
PUT SKIP LIST('ENTER NEXT ITEM:');
```

To inhibit the system prompt for the next GET statement, your own prompt must meet the following conditions:

1. It must be either list-directed or edit-directed, and if list-directed, must be to a PRINT file.
2. The file transmitting the prompt must be allocated to the terminal. If you are merely copying the file at the terminal, the system prompt is not inhibited.

### Using Files Conversationally

TSO allows you to interact conversationally with your own programs, as well as the computing system as a whole. You can perform nearly all your operations from a terminal in TSO: compile PL/I source programs, print the diagnostic messages at the terminal, and write the object modules onto a data set. These object modules can then be conversationally link-edited and run.

While the object modules are running, you can use the terminal as an input and output device for consecutive files in the program. Conversational I/O needs no special PL/I code, so any stream file can be used conversationally.

### Format of Data

The data you enter at the terminal should have exactly the same format as stream input data in batch mode, except for the following variations:

- Simplified punctuation for input: If you enter separate items of input on separate lines, there is no need to enter intervening blanks or commas; the compiler will insert a comma at the end of each line.

For instance, in response to the statement:

```
GET LIST(I,J,K);
```

your terminal interaction could be as follows:

```
:  
1  
+:2  
+:3
```

with a carriage return following each item. It would be equivalent to:

```
:  
1,2,3
```

If you wish to continue an item onto another line, you must end the first line with a continuation character. Otherwise, for a GET LIST or GET DATA statement, a comma will be inserted, and for a GET EDIT statement, the item will be padded (see next paragraph).

- Automatic padding for GET EDIT: There is no need to enter blanks at the end of a line of input for a GET EDIT statement. The item you enter will be padded to the correct length.

For instance, for the PL/I statement:

```
GET EDIT(NAME) (A(15));
```

you could enter the five characters:

```
SMITH
```

followed immediately by a carriage return. The item will be padded with 10 blanks, so that the program receives a string 15 characters long. If you wish to continue an item on a second or subsequent line, you must add a continuation character to the end of every line except the last; the first line transmitted would otherwise be padded and treated as the complete data item.

- SKIP option or format item: A SKIP in a GET statement asks the program to ignore data not yet entered. All uses of SKIP(*n*) where *n* is greater than one are taken to mean SKIP(1). SKIP(1) is taken to mean that all unused data on the current line is ignored.

## Stream and Record Files

You can allocate both stream and record files to the terminal. However, no prompting is provided for record files. If you allocate more than one file to the terminal, and one or more of them is a record file, the output of the files will not necessarily be synchronized. The order in which data is transmitted to and from the terminal is not guaranteed to be the same order in which the corresponding PL/I I/O statements are executed.

Also, record file input from the terminal is received in upper case letters because of a TCAM restriction. To avoid problems you should use stream files wherever possible.

## Capital and Lowercase Letters

For stream files, character strings are transmitted to the program as entered in lowercase or uppercase. For record files, all characters become uppercase.

## End-of-File

The characters /\* in positions one and two of a line that contains no other characters are treated as an end-of-file mark, that is, they raise the ENDFILE condition.

## COPY Option of GET Statement

The GET statement can specify the COPY option; but if the COPY file, as well as the input file, is allocated to the terminal, no copy of the data will be printed.

---

## Controlling Output to the Terminal

At your terminal you can obtain data from a PL/I file that has been both:

1. Declared explicitly or implicitly with the CONSECUTIVE environment option. All stream files meet this condition.
2. Allocated to the terminal.

The standard print file SYSPRINT generally meets both these conditions.

## Format of PRINT Files

Data from SYSPRINT or other PRINT files is not normally formatted into pages at the terminal. Three lines are always skipped for PAGE and LINE options and format items. The ENDPAGE condition is normally never raised. SKIP(*n*), where *n* is greater than three, causes only three lines to be skipped. SKIP(0) is implemented by backspacing, and should therefore not be used with terminals that do not have a backspace feature.

You can cause a PRINT file to be formatted into pages by inserting a tab control table in your program. The table must be called PLITABS, and its contents are explained in “Overriding the Tab Control Table” on page 140. You must initialize the element PAGEDLENGTH to the length of page you require—that is, the length of the sheet of paper on which each page is to be printed, expressed as the maximum number of lines that could be printed on it. You must initialize the element PAGESIZE to the actual number of lines to be printed on each page. After the number of lines in PAGESIZE has been printed on a page, ENDPAGE is raised, for which standard system action is to skip the number of lines equal to PAGEDLENGTH minus PAGESIZE, and then start printing the next page. For other than standard layout, you must initialize the other elements in PLITABS to the values shown in Figure 18 on page 90. You can also use PLITABS to alter the tabulating positions of list-directed and data-directed output. You can use PLITABS for SYSPRINT when you need to format page breaks in ILC applications. Set PAGESIZE to 32767 and use the PUT PAGE statement to control page breaks.

Although some types of terminals have a tabulating facility, tabulating of list-directed and data-directed output is always achieved by transmission of blank characters.

## Stream and Record Files

You can allocate both stream and record files to the terminal. However, if you allocate more than one file to the terminal and one or more is a record file, the files' output will not necessarily be synchronized. There is no guarantee that the order in which data is transmitted between the program and the terminal will be the same as the order in which the corresponding PL/I input and output statements are executed. In addition, because of a TCAM restriction, any output to record files at the terminal is printed in uppercase (capital) letters. It is therefore advisable to use stream files wherever possible.

## Capital and Lowercase Characters

For stream files, characters are displayed at the terminal as they are held in the program, provided the terminal can display them. For instance, with an IBM 327x terminal, capital and lowercase letters are displayed as such, without translation. For record files, all characters are translated to uppercase. A variable or constant in the program can contain lowercase letters if the program was created under the EDIT command with the ASIS operand, or if the program has read lowercase letters from the terminal.

## Output from the PUT EDIT Command

The format of the output from a PUT EDIT command to a terminal has different forms depending on whether the TSO session manager is on or off. Decide whether you want to have it on or off, because if you are using the PUT EDIT command, and change output devices, you will have to rewrite the output procedure. The results of setting TSO session manager on or off are:

- ON** PUT EDIT is converted to full screen TPUTs. The output looks exactly the same as on a disk data set or SYSOUT file.
- OFF** PUT EDIT is converted to line mode TPUTs. "Start of field" and "end of field" characters are added which appear as blanks on the screen.

**Note:** If TSO session manager is not available, format of output will be the same as session manager being off.

## Example of an Interactive Program

The example program in Figure 32 on page 148 prints a report based on information retrieved from a database. The content of the report is controlled by a list of parameters that contains the name of the person requiring the report and a set of numbers indicating the information that is to be printed. In the example, the parameters are read from the terminal. The program includes a prompt for the name parameter, and a message confirming its acceptance. The report is printed on a system output device.

The program uses four files:

- SYSPRINT** Standard stream output file. Prints prompt and confirmation at the terminal.
- PARMS** Stream input file. Reads parameters from terminal.
- INBASE** Record input file. Reads database, namely, member MEM3 of data set BDATA.
- REPORT** Sends report to SYSOUT device.

SYSPRINT has been allocated to the terminal by the logon procedure. The other three files are allocated by ALLOCATE commands entered in TSO submode.

The example program in Figure 32 is called REPORTR, and it is held on a conventionally named TSO data set whose user-supplied name is REPORTER. The compiler is invoked with the SOURCE option to provide a list of the PL/I source code.

---

```

READY
pli reporter print(*) source          "print(*)" allocates
                                       source listing to terminal
15668-910 IBM OS PL/I OPTIMIZING COMPILER VER 2 REL 2 MOD 0
OPTIONS SPECIFIED
S;
                                       SOURCE LISTING
NUMBER
  10 00000010 REPORTR: PROC OPTIONS(MAIN);
      .
      .
  180 00000180 ON ENDFILE(PARMS) GO TO READER;
      .
      .
  1000 00001000 PUT LIST('ENTER NAME:');      print prompt at terminal
  1010 00001010 GET FILE(PARMS) LIST(NAME);    read name parameter from
      .                                       terminal
      .
  1050 00001050 PUT LIST('NAME ACCEPTED');      confirmation message
      .
      .
  2000 00002000 GET FILE(PARMS) LIST((A(I) DO I=1 TO 50));
      .                                       read other parameters
      .                                       from terminal
  2010 00002010 READER:
      00002020 READ FILE(INBASE) INTO(B);
      .                                       read database
      .
  4010 00004010 PRINTER:
      00004020 PUT FILE(REPORT) EDIT(HEAD1||NAME)(A);
      .                                       print line of report
      .                                       on system printer
  5000 00005000 END REPORTR;

NO MESSAGES PRODUCED FOR THIS COMPILATION
COMPILE TIME      0.30 MINS      SPILL FILE: 0 RECORDS, SIZE 4051
END of COMPILATION of REPORTR

READY
alloc file(parms) dataset(*)          file to read parameters
READY                                  from terminal
alloc file(inbase) dataset('bdata(mem3)') old file to read database
READY
alloc file(report) sysout              file to print report on
READY                                  system printer
loadgo reporter plibase

ENTER NAME: 'F W Williams'            prompt & name parameter
NAME ACCEPTED                          confirmation message
:                                       automatic prompt for
                                       parameters
1 3 5 7 10 14 15 19                  parameters entered
+:/*                                  prompt for further
                                       parameters
READY                                  end-of-file entered

```

---

Figure 32. Example of an Interactive Program

## Using Record-Oriented Data Transmission

PL/I supports various types of data sets with the RECORD attribute (see Table 23 on page 156). This section covers how to use consecutive data sets.

Table 20 lists the statements and options that you can use to create and access a consecutive data set using record-oriented data transmission.

*Table 20. Statements and Options Allowed for Creating and Accessing Consecutive Data Sets*

<b>File declaration<sup>1</sup></b>	<b>Valid statements,<sup>2</sup> with Options you must specify</b>	<b>Other options you can specify</b>
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference);	
	LOCATE based-variable FILE(file-reference);	SET(pointer-reference)
SEQUENTIAL OUTPUT UNBUFFERED	WRITE FILE(file-reference) FROM(reference);	EVENT(event-reference)
SEQUENTIAL INPUT BUFFERED <sup>3</sup>	READ FILE(file-reference) INTO(reference);	
	READ FILE(file-reference) SET(pointer-reference);	
	READ FILE(file-reference) IGNORE(expression);	
SEQUENTIAL INPUT UNBUFFERED <sup>3</sup>	READ FILE(file-reference) INPUT(reference);	EVENT(event-reference)
	READ FILE(file-reference) IGNORE(expression);	EVENT(event-reference)
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);	
	READ FILE(file-reference) SET(pointer-reference);	
	READ FILE(file-reference) IGNORE(expression);	
	REWRITE FILE(file-reference);	FROM(reference)
SEQUENTIAL UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference);	EVENT(event-reference)
	READ FILE(file-reference) IGNORE(expression);	EVENT(event-reference)
	REWRITE FILE(file-reference) FROM(reference);	EVENT(event-reference)

**Notes:**

1. The complete file declaration would include the attributes FILE, RECORD and ENVIRONMENT.
2. The statement READ FILE (file-reference); is a valid statement and is equivalent to READ FILE(file-reference) IGNORE (1);
3. You can specify the BACKWARDS attribute for files on magnetic tape.

## Using Magnetic Tape without Standard Labels

If a magnetic-tape data set has nonstandard labels or is unlabeled, you must specify the block size either in your PL/I program (ENVIRONMENT attribute) or in the DD statement (BLKSIZE subparameter). The DSNNAME parameter is not essential if the data set is not cataloged.

PL/I includes no facilities for processing nonstandard labels which to the operating system appear as data sets preceding or following your data set. You can either process the labels as independent data sets or use the LABEL parameter of the DD statement to bypass them. To bypass the labels, code LABEL=(2,NL) or LABEL=(,BLP).

## Specifying Record Format

If you give record-format information, it must be compatible with the actual structure of the data set. For example, if you create a data set with FB-format records, with a record size of 600 bytes and a block size of 3600 bytes, you can access the records as if they are U-format with a maximum block size of 3600 bytes. If you specify a block size of 3500 bytes, your data is truncated.

## Defining Files Using Record I/O

You define files for record-oriented data transmission by using a file declaration with the following attributes:

```
DCL filename FILE RECORD
      INPUT | OUTPUT | UPDATE
      SEQUENTIAL
      BUFFERED | UNBUFFERED
      [BACKWARDS]
      ENVIRONMENT(options);
```

Default file attributes are shown in Table 15 on page 111. The file attributes are described in the *PL/I for MVS & VM Language Reference*. Options of the ENVIRONMENT attribute are discussed below.

## Specifying ENVIRONMENT Options

The ENVIRONMENT options applicable to consecutive data sets are:

```
F|FB|FS|FBS|V|VB|VS|VBS|D|DB|U
RECSIZE(record-length)
BLKSIZE(block-size)
SCALARVARYING
COBOL
BUFFERS(n)
NCP(n)
TRKOFL

CONSECUTIVE
TOTAL
CTLASA|CTL360
LEAVE|REREAD
ASCII
BUFOFF[(n)]
```

The options above the blank line are described in “Specifying Characteristics in the ENVIRONMENT Attribute” on page 110, and those below the blank line are described below. D- and DB-format records are also described below.

See Table 15 on page 111 to find which options you must specify, which are optional, and which are defaults.

## **CONSECUTIVE**

The **CONSECUTIVE** option defines a file with consecutive data set organization, which is described in this chapter and in “Data Set Organization” on page 106.

►—CONSECUTIVE—◄

**CONSECUTIVE** is the default when the merged attributes from the **DECLARE** and **OPEN** statements do not include the **TRANSIENT** attribute.

## **TOTAL**

In general, run-time library subroutines called from object code perform I/O operations. Under certain conditions, however, the compiler can, when requested, provide in-line code to carry out these operations. This gives faster execution of the I/O statements.

Use the **TOTAL** option to aid the compiler in the production of efficient object code. In particular, it requests the compiler to use in-line code for certain I/O operations. It specifies that no attributes will be merged from the **OPEN** statement or the I/O statement or the **DCB** parameter; if a complete set of attributes can be built up at compile time from explicitly declared and default attributes, in-line code will be used for certain I/O operations.

►—TOTAL—◄

The **UNDEFINEDFILE** condition is raised if any attribute that was not explicitly declared appears on the **OPEN** statement, or if the I/O statement implies a file attribute that conflicts with a declared or default attribute.

You cannot specify the **TOTAL** option for device-associated files or files reading Optical Mark Read data.

The use of in-line I/O code can result in reduced error-handling capability. In particular, if a program-check interrupt or an abend occurs during in-line I/O, the error message produced can contain incorrect offset and statement number information. Also, execution of a **GO TO** statement in an **ERROR ON-unit** for such an interrupt can cause a second program check.

There are some differences in the optimized code generated under OS PL/I Version 1 Release 5 and later releases. The implementation of these releases generates code to call modules in the run-time library so that mode-switching can be performed if necessary. This implementation results in a longer instruction path than it does with prior releases, but it is still faster than not using the **TOTAL** option.

Table 21 on page 152 shows the conditions under which I/O statements are handled in-line.

When in-line code is employed to implement an I/O statement, the compiler gives an informational message.

Table 21. Conditions under Which I/O Statements Are Handled In-Line (TOTAL Option Used)

Statement <sup>1</sup>	Record variable requirements	File attribute <sup>3</sup> or ENVIRONMENT option requirements <sup>4</sup>
READ SET	None	Not BACKWARDS for record types U, V, VB
READ INTO	Length known at compile time, maximum length for a varying string or area. <sup>2</sup>	RECSIZE known at compile time. <sup>5</sup> SCALARVARYING option if varying string.
WRITE FROM (fixed string)	Length known at compile time.	RECSIZE known at compile time. <sup>5</sup>
WRITE FROM (varying string)		RECSIZE known at compile time. <sup>5</sup> SCALARVARYING option used.
WRITE FROM Area <sup>2</sup>		RECSIZE known at compile time. <sup>5</sup>
LOCATE A	Length known at compile time, maximum length for a varying string or area. <sup>2</sup>	RECSIZE known at compile time. <sup>5</sup> SCALARVARYING if varying string.

**Notes:**

1. All statements must be found to be valid during compilation. File parameters or file variables are *never* handled by in-line code.
2. Including structures wherein the last element is an unsubscripted area.
3. File attributes are SEQUENTIAL BUFFERED, INPUT, or OUTPUT.
4. Data set organization must be CONSECUTIVE; allowable record formats are F, FB, FS, FBS, U, V, or VB.
5. You can specify BLKSIZE instead of RECSIZE for unblocked record formats F, FS, V, and U.

## CTLASA|CTL360

The printer control options CTLASA and CTL360 apply only to OUTPUT files associated with consecutive data sets. They specify that the first character of a record is to be interpreted as a control character.



The CTLASA option specifies American National Standard Vertical Carriage Positioning Characters or American National Standard Pocket Select Characters (Level 1). The CTL360 option specifies IBM machine-code control characters.

The American National Standard control characters, listed in Figure 33 on page 153, cause the specified action to occur before the associated record is printed or punched.

The machine code control characters differ according to the type of device. The IBM machine code control characters for printers are listed in Figure 34 on page 153.

---

Code	Action
	Space 1 line before printing (blank code)
0	Space 2 lines before printing
-	Space 3 lines before printing
+	Suppress space before printing
1	Skip to channel 1
2	Skip to channel 2
3	Skip to channel 3
4	Skip to channel 4
5	Skip to channel 5
6	Skip to channel 6
7	Skip to channel 7
8	Skip to channel 8
9	Skip to channel 9
A	Skip to channel 10
B	Skip to channel 11
C	Skip to channel 12
V	Select stacker 1
W	Select stacker 2

---

Figure 33. American National Standard Print and Card Punch Control Characters (CTLASA)

---

Print and Then Act	Action	Act immediately (no printing)
<b>Code byte</b>		<b>Code byte</b>
00000001	Print only (no space)	—
00001001	Space 1 line	00001011
00010001	Space 2 lines	00010011
00011001	Space 3 lines	00011011
10001001	Skip to channel 1	10001011
10010001	Skip to channel 2	10010011
10011001	Skip to channel 3	10011011
10100001	Skip to channel 4	10100011
10101001	Skip to channel 5	10101011
10110001	Skip to channel 6	10110011
10111001	Skip to channel 7	10111011
11000001	Skip to channel 8	11000011
11001001	Skip to channel 9	11001011
11010001	Skip to channel 10	11010011
11011001	Skip to channel 11	11011011
11100001	Skip to channel 12	11100011

---

Figure 34. IBM Machine Code Print Control Characters (CTL360)

## LEAVE|REREAD

The magnetic tape handling options LEAVE and REREAD allow you to specify the action to be taken when the end of a magnetic tape volume is reached, or when a data set on a magnetic tape volume is closed. The LEAVE option prevents the tape from being rewound. The REREAD option rewinds the tape to allow reprocessing of the data set. If you do not specify either of these, the action at end-of-volume or on closing of a data set is controlled by the DISP parameter of the associated DD statement.



If a data set is first read or written forward and then read backward in the same program, specify the LEAVE option to prevent rewinding when the file is closed (or, with a multivolume data set, when volume switching occurs).

You can also specify LEAVE and REREAD on the CLOSE statement, as described in the *PL/I for MVS & VM Language Reference*.

The effects of the LEAVE and REREAD options are summarized in Table 22.

Table 22. Effect of LEAVE and REREAD Options

ENVIRONMENT option	DISP parameter	Action
REREAD	—	Positions the current volume to reprocess the data set. Repositioning for a BACKWARDS file is at the physical end of the data set.
LEAVE	—	Positions the current volume at the logical end of the data set. Repositioning for a BACKWARDS file is at the physical beginning of the data set.
Neither REREAD nor LEAVE	PASS	Positions the volume at the end of the data set.
	DELETE	Rewinds the current volume.
	KEEP, CATLG, UNCATLG	Rewinds and unloads the current volume.

## ASCII

The ASCII option specifies that the code used to represent data on the data set is ASCII.

►—ASCII—◄

You can create and access data sets on magnetic tape using ASCII in PL/I. The implementation supports F, FB, U, D, and DB record formats. F, FB, and U formats are treated in the same way as other data sets; D and DB formats, which correspond to V and VB formats in other data sets, are described below.

Only character data can be written to an ASCII data set; therefore, when you create the data set, you must transmit your data from character variables. You can give these variables the attribute VARYING as well as CHARACTER, but you cannot transmit the two length bytes of varying-length character strings. In other words, you cannot use a SCALARVARYING file to transmit varying-length character strings to an ASCII data set. Also, you cannot transmit data aggregates containing varying-length strings.

Since an ASCII data set must be on magnetic tape, it must be of consecutive organization. The associated file must be BUFFERED. You can also specify the BUFOFF ENVIRONMENT option for ASCII data sets.

If you do not specify ASCII in either the ENVIRONMENT option or the DD statement, but you specify BUFOFF, D, or DB, then ASCII is the default.

## BUFOFF

You need not concern yourself with the BUFOFF option unless you are dealing with ASCII data sets.

The BUFOFF (buffer offset) option specifies a *block prefix* field  $n$  bytes in length at the beginning of each block in an ASCII data set, according to the following syntax:



$n$  is either:

- An integer from 0 to 99
- A variable with attributes FIXED BINARY(31,0) STATIC having an integer value from 0 to 99.

When you are accessing an ASCII data set for input to your program, specifying BUFOFF and  $n$  identifies to data management how far into the block the beginning of the data is. Specifying BUFOFF without  $n$  signifies to data management that the first 4 bytes of the data set comprise a block-length field.

When you are creating an ASCII data set for output from your program, PL/I does not allow you to create a prefix field at the beginning of the block using BUFOFF, *unless* it is for data management's use as a 4-byte block-length indicator. In this case, you do not need to specify the BUFOFF option anyway, because for D- or DB-formats PL/I automatically sets up the required field. You *can* code BUFOFF without  $n$  (though it isn't needed), but that is the only explicit specification of the BUFOFF option that PL/I accepts for output. Therefore, by not coding the BUFOFF option you allow PL/I to set the default values needed for creating your output ASCII data set (4 for D- and DB-formats, 0 for other acceptable formats).

## D-Format and DB-Format Records

The data contained in D- and DB-format records is recorded in ASCII. Each record can be of a different length. The two formats are:

### D-format:

The records are unblocked; each record constitutes a single block. Each record consists of:

Four control bytes  
Data bytes.

The four control bytes contain the length of the record; this value is inserted by data management and requires no action by you. In addition, there can be, at the start of the block, a block prefix field, which can contain the length of the block.

### DB-format:

The records are blocked. All other information given for D-format applies to DB-format.

## Creating a Data Set with Record I/O

When you create a consecutive data set, you must open the associated file for SEQUENTIAL OUTPUT. You can use either the WRITE or LOCATE statement to write records. Table 20 on page 149 shows the statements and options for creating a consecutive data set.

When creating a data set, you must identify it to the operating system in a DD statement. The following paragraphs, summarized in Table 23, tell what essential information you must include in the DD statement and discuss some of the optional information you can supply.

*Table 23. Creating a Consecutive Data Set with Record I/O: Essential Parameters of the DD Statement*

Storage device	When required	What you must state	Parameters
All	Always	Output device  Block size <sup>1</sup>	UNIT= or SYSOUT= or VOLUME=REF=  DCB=(BLKSIZE=...
Direct access only	Always	Storage space required	SPACE=
Magnetic tape only	Data set not first in volume and for magnetic tapes that do not have standard labels	Sequence number	LABEL=
Direct access and standard labeled magnetic tape	Data set to be used by another job step but not required at end of job	Disposition	DISP=
	Data set to be kept after end of job	Disposition	DISP=
	Data set to be on particular device	Name of data set  Volume serial number	DSNAME=  VOLUME=SER= or VOLUME=REF=

<sup>1</sup>Or you could specify the block size in your PL/I program by using the ENVIRONMENT attribute.

### Essential Information

When you create a consecutive data set you must specify:

- The device that will write your data set (UNIT, SYSOUT, or VOLUME parameter of DD statement): A data set with consecutive organization can exist on any type of auxiliary storage device.
- The block size: You can specify the block size either in your PL/I program (ENVIRONMENT attribute) or in the DD statement (BLKSIZE subparameter). If you do not specify a record length, unblocked records are the default and the record length is determined from the block size. If you do not specify a record format, U-format is the default. If you specify a record size and either specify a block size of zero or omit a specification for it, under MVS/ESA, DFP calculates a block size.

If you want to keep a magnetic-tape or direct-access data set (that is, you do not want the operating system to delete it at the end of your job), the DD statement must name the data set and indicate how it is to be disposed of (DSNAME and DISP parameters). The DISP parameter alone will suffice if you want to use the data set in a later step but will not need it after the end of your job.

When creating a data set on a direct-access device, you must specify the amount of space required for it (SPACE parameter of DD statement).

If you want your data set stored on a particular magnetic-tape or direct-access device, you must specify the volume serial number in the DD statement (SER or REF subparameter of VOLUME parameter). If you do not specify a serial number for a magnetic-tape data set that you want to keep, the operating system will allocate one, inform the operator, and print the number on your program listing.

If your data set is to follow another data set on a magnetic-tape volume, you must use the LABEL parameter of the DD statement to indicate its sequence number on the tape.

The DCB subparameters of the DD statement that apply to consecutive data sets are listed below. They are described in your *MVS/ESA JCL User's Guide*. Table 15 on page 111 shows which options of the ENVIRONMENT attribute you can specify for consecutive data sets.

#### **Subparameter Specifies**

<b>BLKSIZE</b>	Maximum number of bytes per block
<b>BUFNO</b>	Number of data management buffers
<b>CODE</b>	Paper tape: code in which the tape is punched
<b>DEN</b>	Magnetic tape: tape recording density
<b>FUNC</b>	Card reader or punch: function to be performed
<b>LRECL</b>	Maximum number of bytes per record
<b>MODE</b>	Card reader or punch: mode or operation (column binary or EBCDIC and Read Column Eliminate or Optical Mark Read)
<b>OPTCD</b>	Optional data-management services and data-set attributes
<b>PRTSP</b>	Printer line spacing (0, 1, 2, or 3)
<b>RECFM</b>	Record format and characteristics
<b>STACK</b>	Card reader or punch: stacker selection
<b>TRTCH</b>	Magnetic tape: tape recording technique for 7-track tape

## **Accessing and Updating a Data Set with Record I/O**

Once you create a consecutive data set, you can open the file that accesses it for sequential input, for sequential output, or, for data sets on direct-access devices, for updating. See Figure 35 on page 160 for an example of a program that accesses and updates a consecutive data set. If you open the file for output, and extend the data set by adding records at the end, you must specify DISP=MOD in the DD statement. If you do not, the data set will be overwritten. If you open a file for updating, you can only update records in their existing sequence, and if you want to insert records, you must create a new data set. Table 20 on page 149 shows the statements and options for accessing and updating a consecutive data set.

When you access a consecutive data set by a SEQUENTIAL UPDATE file, you must retrieve a record with a READ statement before you can update it with a REWRITE statement; however, every record that is retrieved need not be rewritten. A REWRITE statement will always update the last record read.

Consider the following:

```

READ FILE(F) INTO(A);
.
.
.
READ FILE(F) INTO(B);
.
.
.
REWRITE FILE(F) FROM(A);

```

The REWRITE statement updates the record that was read by the second READ statement. The record that was read by the first statement cannot be rewritten after the second READ statement has been executed.

The operating system does not allow updating a consecutive data set on magnetic tape except by adding records at the end. To replace or insert records, you must read the data set and write the updated records into a new data set.

You can read a consecutive data set on magnetic tape forward or backward. If you want to read the data set backward, you must give the associated file the BACKWARDS attribute. You cannot specify the BACKWARDS attribute when a data set has V-, VB-, VS-, VBS-, D-, or DB-format records.

To access a data set, you must identify it to the operating system in a DD statement. Table 24 summarizes the DD statement parameters needed to access a consecutive data set.

*Table 24. Accessing a Consecutive Data Set with Record I/O: Essential Parameters of the DD Statement*

When required	What you must state	Parameters
Always	Name of data set	DSNAME=
	Disposition of data set	DISP=
If data set not cataloged (all devices)	Input device	UNIT= or VOLUME=REF=
If data set not cataloged (standard labeled magnetic tape and direct access)	Volume serial number	VOLUME=SER=
Magnetic tape (if data set not first in volume or which does not have standard labels)	Sequence number	LABEL=
If data set does not have standard labels	Block size <sup>1</sup>	DCB=(BLKSIZE=.

<sup>1</sup>Or you could specify the block size in your PL/I program by using the ENVIRONMENT attribute.

The following paragraphs indicate the essential information you must include in the DD statement, and discuss some of the optional information you can supply. The discussions do not apply to data sets in the input stream.

## Essential Information

If the data set is cataloged, you need to supply only the following information in the DD statement:

- The name of the data set (DSNAME parameter). The operating system will locate the information describing the data set in the system catalog, and, if necessary, will request the operator to mount the volume containing it.
- Confirmation that the data set exists (DISP parameter). If you open the data set for output with the intention of extending it by adding records at the end, code DISP=MOD; otherwise, opening the data set for output will result in it being overwritten.

If the data set is not cataloged, you must, in addition, specify the device that will read the data set and, for magnetic-tape and direct-access devices, give the serial number of the volume that contains the data set (UNIT and VOLUME parameters).

If the data set follows another data set on a magnetic-tape volume, you must use the LABEL parameter of the DD statement to indicate its sequence number on the tape.

## Example of Consecutive Data Sets

Creating and accessing consecutive data sets are illustrated in the program in Figure 35 on page 160. The program merges the contents of two data sets, in the input stream, and writes them onto a new data set, &&TEMP; each of the original data sets contains 15-byte fixed-length records arranged in EBCDIC collating sequence. The two input files, INPUT1 and INPUT2, have the default attribute BUFFERED, and locate mode is used to read records from the associated data sets into the respective buffers. Access of based variables in the buffers should not be attempted after the file has been closed; in MVS/XA DFP has released the buffer, and a protection error might result.

---

```

//EXAMPLE JOB
//STEP1 EXEC IEL1CLG
//PLI.SYSIN DD *
%PROCESS INT F(I) AG A(F) ESD MAP OP STG NEST X(F) SOURCE ;
%PROCESS LIST;

MERGE: PROC OPTIONS(MAIN);
  DCL (INPUT1,                               /* FIRST INPUT FILE */
      INPUT2,                               /* SECOND INPUT FILE */
      OUT ) FILE RECORD SEQUENTIAL; /* RESULTING MERGED FILE*/
  DCL SYSPRINT FILE PRINT; /* NORMAL PRINT FILE */

  DCL INPUT1_EOF BIT(1) INIT('0'B); /* EOF FLAG FOR INPUT1 */
  DCL INPUT2_EOF BIT(1) INIT('0'B); /* EOF FLAG FOR INPUT2 */
  DCL OUT_EOF BIT(1) INIT('0'B); /* EOF FLAG FOR OUT */
  DCL TRUE BIT(1) INIT('1'B); /* CONSTANT TRUE */
  DCL FALSE BIT(1) INIT('0'B); /* CONSTANT FALSE */

  DCL ITEM1 CHAR(15) BASED(A); /* ITEM FROM INPUT1 */
  DCL ITEM2 CHAR(15) BASED(B); /* ITEM FROM INPUT2 */
  DCL INPUT_LINE CHAR(15); /* INPUT FOR READ INTO */
  DCL A POINTER; /* POINTER VAR */
  DCL B POINTER; /* POINTER VAR */

  ON ENDFILE(INPUT1) INPUT1_EOF = TRUE;
  ON ENDFILE(INPUT2) INPUT2_EOF = TRUE;
  ON ENDFILE(OUT) OUT_EOF = TRUE;

  OPEN FILE(INPUT1) INPUT,
      FILE(INPUT2) INPUT,
      FILE(OUT) OUTPUT;

  READ FILE(INPUT1) SET(A); /* PRIMING READ */
  READ FILE(INPUT2) SET(B);

  DO WHILE ((INPUT1_EOF = FALSE) & (INPUT2_EOF = FALSE));
  IF ITEM1 > ITEM2 THEN
  DO;
  WRITE FILE(OUT) FROM(ITEM2);
  PUT FILE(SYSPRINT) SKIP EDIT('1>2', ITEM1, ITEM2)
  (A(5),A,A);
  READ FILE(INPUT2) SET(B);
  END;
  ELSE
  DO;
  WRITE FILE(OUT) FROM(ITEM1);
  PUT FILE(SYSPRINT) SKIP EDIT('1<2', ITEM1, ITEM2)
  (A(5),A,A);
  READ FILE(INPUT1) SET(A);
  END;
  END;

```

---

Figure 35 (Part 1 of 2). Merge Sort—Creating and Accessing a Consecutive Data Set

---

```

DO WHILE (INPUT1_EOF = FALSE);          /* INPUT2 IS EXHAUSTED */
  WRITE FILE(OUT) FROM(ITEM1);
  PUT FILE(SYSPRINT) SKIP EDIT('1', ITEM1) (A(2),A);
  READ FILE(INPUT1) SET(A);
END;

DO WHILE (INPUT2_EOF = FALSE);          /* INPUT1 IS EXHAUSTED */
  WRITE FILE(OUT) FROM(ITEM2);
  PUT FILE(SYSPRINT) SKIP EDIT('2', ITEM2) (A(2),A);
  READ FILE(INPUT2) SET(B);
END;

CLOSE FILE(INPUT1), FILE(INPUT2), FILE(OUT);
PUT FILE(SYSPRINT) PAGE;
OPEN FILE(OUT) SEQUENTIAL INPUT;

READ FILE(OUT) INTO(INPUT_LINE);        /* DISPLAY OUT FILE */
DO WHILE (OUT_EOF = FALSE);
  PUT FILE(SYSPRINT) SKIP EDIT(INPUT_LINE) (A);
  READ FILE(OUT) INTO(INPUT_LINE);
END;
CLOSE FILE(OUT);

END MERGE;
/*
//GO.INPUT1 DD *
AAAAAA
CCCCCC
EEEEEE
GGGGGG
IIIIII
/*
//GO.INPUT2 DD *
BBBBBB
DDDDDD
FFFFFF
HHHHHH
JJJJJJ
KKKKKK
/*
//GO.OUT DD DSN=&&TEMP,DISP=(NEW,DELETE),UNIT=SYSDA,
//          DCB=(RECFM=FB,BLKSIZE=150,LRECL=15),SPACE=(TRK,(1,1))

```

---

*Figure 35 (Part 2 of 2). Merge Sort—Creating and Accessing a Consecutive Data Set*

The program in Figure 36 on page 162 uses record-oriented data transmission to print the table created by the program in Figure 30 on page 141.

---

```

%PROCESS INT F(I) AG A(F) ESD MAP OP STG NEST X(F) SOURCE ;
%PROCESS LIST;

PRT: PROC OPTIONS(MAIN);
  DCL TABLE      FILE RECORD INPUT SEQUENTIAL;
  DCL PRINTER     FILE RECORD OUTPUT SEQL
                  ENV(V BLKSIZE(102) CTLASA);
  DCL LINE        CHAR(94) VAR;

  DCL TABLE_EOF  BIT(1) INIT('0'B);      /* EOF FLAG FOR TABLE */
  DCL TRUE        BIT(1) INIT('1'B);      /* CONSTANT TRUE       */
  DCL FALSE       BIT(1) INIT('0'B);      /* CONSTANT FALSE      */

  ON ENDFILE(TABLE) TABLE_EOF = TRUE;

  OPEN FILE(TABLE),
        FILE(PRINTER);

  READ FILE(TABLE) INTO(LINE);             /* PRIMING READ        */

  DO WHILE (TABLE_EOF = FALSE);
    WRITE FILE(PRINTER) FROM(LINE);
    READ FILE(TABLE) INTO(LINE);
  END;

  CLOSE FILE(TABLE),
        FILE(PRINTER);
END PRT;

```

---

*Figure 36. Printing Record-Oriented Data Transmission*

---

## Chapter 9. Defining and Using Indexed Data Sets

This chapter describes indexed data set organization (ISAM), data transmission statements, and ENVIRONMENT options that define indexed data sets. It then describes how to create, access, and reorganize indexed data sets. Use of ISAM is discouraged for new data sets because VSAM gives better performance with PL/I. ISAM is retained for compatibility with existing data sets.

Under VM, PL/I supports the use of Indexed Data Sets through VSAM. See “Using Data Sets and Files” on page 81 for more information on VSAM data sets under VM.

### Indexed Organization

A data set with indexed organization must be on a direct-access device. Its records can be either F-format or V-format records, blocked or unblocked. The records are arranged in logical sequence, according to keys associated with each record. A *key* is a character string that can identify each record uniquely. Logical records are arranged in the data set in ascending key sequence according to the EBCDIC collating sequence. Indexes associated with the data set are used by the operating system data-management routines to locate a record when the key is supplied.

Unlike consecutive organization, indexed organization does not require you to access every record in sequential fashion. You must create an indexed data set sequentially; but once you create it, you can open the associated file for SEQUENTIAL or DIRECT access, as well as INPUT or UPDATE. When the file has the DIRECT attribute, you can retrieve, add, delete, and replace records at random.

Sequential processing of an indexed data set is slower than that of a corresponding consecutive data set, because the records it contains are not necessarily retrieved in physical sequence. Furthermore, random access is less efficient for an indexed data set than for a regional data set, because the indexes must be searched to locate a record. An indexed data set requires more external storage space than a consecutive data set, and all volumes of a multivolume data set must be mounted, even for sequential processing.

Table 25 on page 164 lists the data-transmission statements and options that you can use to create and access an indexed data set.

### Using keys

There are two kinds of keys—recorded keys and source keys. A *recorded key* is a character string that actually appears with each record in the data set to identify that record. The length of the recorded key cannot exceed 255 characters and all keys in a data set must have the same length. The recorded keys in an indexed data set can be separate from, or embedded within, the logical records. A *source key* is the character value of the expression that appears in the KEY or KEYFROM option of a data transmission statement to identify the record to which the statement refers. For direct access of an indexed data set, you must include a source key in each transmission statement.

**Note:** All VSAM key-sequenced data sets have embedded keys, even if they have been converted from ISAM data sets with nonembedded keys.

*Table 25 (Page 1 of 2). Statements and Options Allowed for Creating and Accessing Indexed Data Sets*

<b>File declaration<sup>1</sup></b>	<b>Valid statements, with options you must include</b>	<b>Other options you can include</b>
SEQUENTIAL OUTPUT	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
	LOCATE based-variable FILE(file-reference) KEYFROM(expression);	SET(pointer-reference)
SEQUENTIAL INPUT	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) IGNORE(expression);	
SEQUENTIAL UPDATE	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) IGNORE(expression);	
	REWRITE FILE(file-reference);	FROM(reference)
	DELETE FILE(file-reference); <sup>2</sup>	KEY(expression)
DIRECT INPUT	READ FILE(file-reference) INTO(reference) KEY(expression);	EVENT(event-reference)
DIRECT UPDATE	READ FILE(file-reference) INTO(reference) KEY(expression);	EVENT(event-reference)
	REWRITE FILE(file-reference) FROM(reference) KEY(expression);	EVENT(event-reference)
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	EVENT(event-reference)
	DELETE FILE(file-reference) KEY(expression); <sup>2</sup>	EVENT(event-reference)

Table 25 (Page 2 of 2). Statements and Options Allowed for Creating and Accessing Indexed Data Sets

File declaration <sup>1</sup>	Valid statements, with options you must include	Other options you can include
DIRECT UPDATE EXCLUSIVE	READ FILE(file-reference) INTO(reference) KEY(expression);	EVENT(event-reference) and/or NOLOCK
	REWRITE FILE(file-reference) FROM(reference) KEY(expression);	EVENT(event-reference)
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	EVENT(event-reference)
	DELETE FILE(file-reference) KEY(expression); <sup>2</sup>	EVENT(event-reference)
	UNLOCK FILE(file-reference) KEY(expression)	

**Notes:**

1. The complete file declaration would include the attributes FILE, RECORD, and ENVIRONMENT. If you use any of the options KEY, KEYFROM, or KEYTO, you must also include the attribute KEYED in the file declaration. The attribute BUFFERED is the default, and UNBUFFERED is ignored for INDEXED SEQUENTIAL and SEQUENTIAL files.
2. Use of the DELETE statement is invalid if you did not specify OPTCD=L (DCB subparameter) when the data set was created or if the RKP subparameter is 0 for FB records, or 4 for V and VB records.

The use of embedded keys avoids the need for the KEYTO option during sequential input, but the KEYFROM option is still required for output. (However, the data specified by the KEYFROM option can be the embedded key portion of the record variable itself.) In a data set with unblocked records, a separate recorded key precedes each record, even when there is already an embedded key. If the records are blocked, the key of only the last record in each block is recorded separately in front of the block.

During execution of a WRITE statement that adds a record to a data set with embedded keys, the value of the expression in the KEYFROM option is assigned to the embedded key position in the record variable. Note that you can declare a record variable as a structure with an embedded key declared as a structure member, but that you must not declare such an embedded key as a VARYING string.

For a REWRITE statement using SEQUENTIAL files with indexed data set organization, you must ensure that the rewritten key is the same as the key in the replaced record.

For a LOCATE statement, the KEYFROM string is assigned to the embedded key when the next operation on the file is encountered.

## Using Indexes

To provide faster access to the records in the data set, the operating system creates and maintains a system of indexes to the records in the data set.

The lowest level of index is the *track index*. There is a track index for each cylinder in the data set. The track index occupies the first track (or tracks) of the cylinder, and lists the key of the last record on each track in the cylinder. A search can then be directed to the first track that has a key that is higher than or equal to the key of the required record.

If the data set occupies more than one cylinder, the operating system develops a higher-level index called a *cylinder index*. Each entry in the cylinder index identifies the key of the last record in the cylinder.

To increase the speed of searching the cylinder index, you can request in a DD statement that the operating system develop a *master index* for a specified number of cylinders. You can have up to three levels of master index.

Figure 37 illustrates the index structure. The part of the data set that contains the cylinder and master indexes is termed the *index area*.

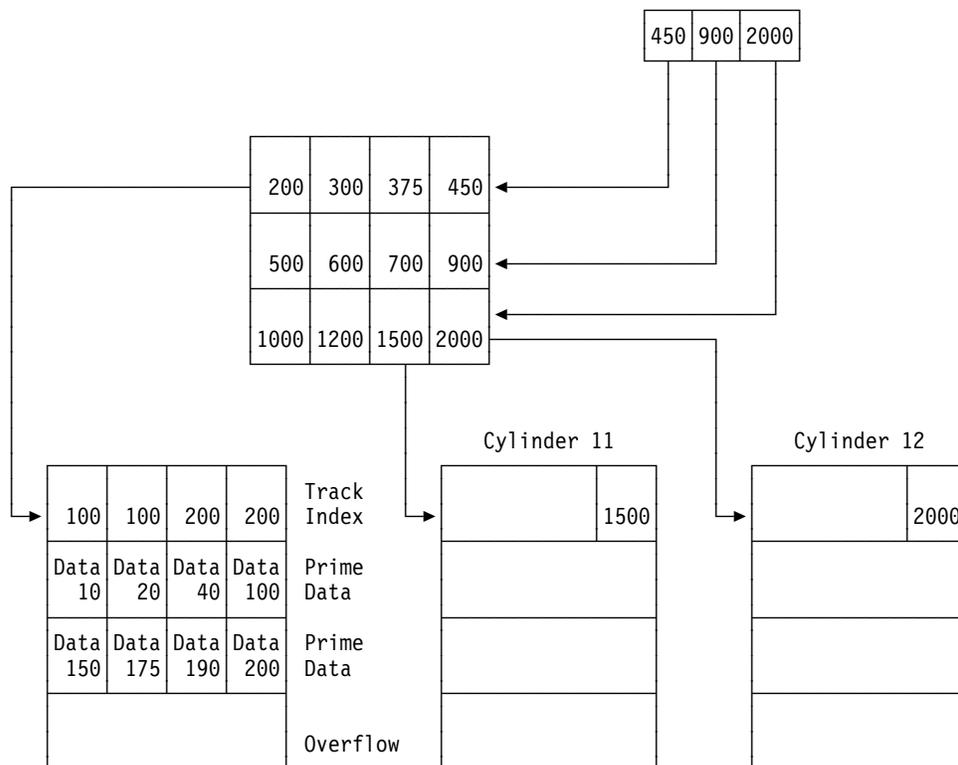


Figure 37. Index Structure of an Indexed Data Set

When you create an indexed data set, all the records are written in what is called the *prime data area*. If you add more records later, the operating system does not rearrange the entire data set; it inserts each new record in the appropriate position and moves up the other records on the same track. Any records forced off the track by the insertion of a new record are placed in an *overflow area*. The overflow area can be either a number of tracks set aside in each cylinder for the overflow records from that cylinder (*cylinder overflow area*), or a separate area for all overflow records (*independent overflow area*).

Records in the overflow area are chained together to the track index so as to maintain the logical sequence of the data set. This is illustrated in Figure 38 on page 168. Each entry in the track index consists of two parts:

- The normal entry, which points to the last record on the track
- The overflow entry, which contains the key of the first record transferred to the overflow area and also points to the last record transferred from the track to the overflow area.

If there are no overflow records from the track, both index entries point to the last record on the track. An additional field is added to each record that is placed in the overflow area. It points to the previous record transferred from the same track. The first record from each track is linked to the corresponding overflow entry in the track index.

### **Dummy Records**

Records within an indexed data set are either actual records, containing valid data, or dummy records. A dummy record, identified by the constant (8)'1'B in its first byte, can be one that you insert or it can be created by the operating system. You insert dummy records by setting the first byte to (8)'1'B and writing the records in the usual way. The operating system creates dummy records by placing (8)'1'B in a record that is named in a DELETE statement.

When creating an indexed data set, you might want to insert dummy records to reserve space in the prime data area. You can replace dummy records later with actual data records having the same key.

The operating system removes dummy records when the data set is reorganized, as described later in this section, and removes those forced off the track during an update.

If you include the DCB subparameter OPTCD=L in the DD statement that defines the data set when you create it, dummy records will not be retrieved by READ statements and the operating system will write the dummy identifier in records being deleted.

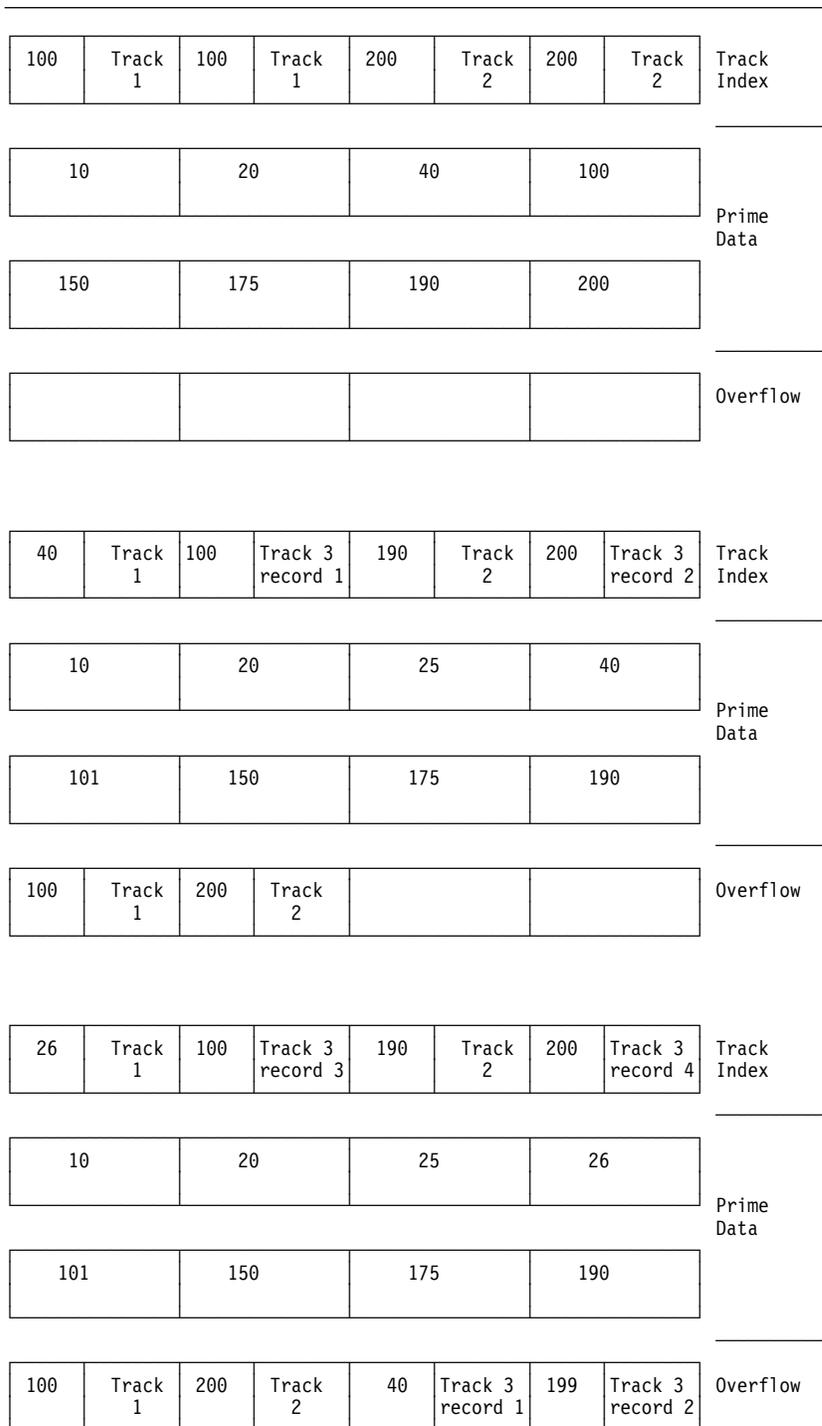


Figure 38. Adding Records to an Indexed Data Set

---

## Defining Files for an Indexed Data Set

You define a sequential indexed data set by a file declaration with the following attributes:

```
DCL filename FILE RECORD
      INPUT | OUTPUT | UPDATE
      SEQUENTIAL
      BUFFERED
      [KEYED]
      ENVIRONMENT(options);
```

You define a direct indexed data set by a file declaration with the following attributes:

```
DCL filename FILE RECORD
      INPUT | OUTPUT | UPDATE
      DIRECT
      UNBUFFERED
      KEYED
      [EXCLUSIVE]
      ENVIRONMENT(options);
```

Default file attributes are shown in Table 15 on page 111. The file attributes are described in the *PL/I for MVS & VM Language Reference*. Options of the ENVIRONMENT attribute are discussed below.

## Specifying ENVIRONMENT Options

The ENVIRONMENT options applicable to indexed data sets are:

```
F|FB|V|VB
RECSIZE(record-length)
BLKSIZE(block-size)
SCALARVARYING
COBOL
BUFFERS(n)
KEYLENGTH(n)
NCP(n)
GENKEY

ADDBUFF
INDEXAREA[(index-area-size)]
INDEXED
KEYLOC(n)
NOWRITE
```

The options above the blank line are described in “Specifying Characteristics in the ENVIRONMENT Attribute” on page 110, and those below the blank line are described below.

### ADDBUFF Option

Specify the ADDBUFF option for a DIRECT INPUT or DIRECT UPDATE file with indexed data set organization and F-format records to indicate that an area of internal storage is used as a workspace in which records on the data set can be rearranged when new records are added. The size of the workspace is equivalent to one track of the direct-access device used.

You do not need to specify the ADDBUFF option for DIRECT INDEXED files with V-format records, as the workspace is automatically allocated for such files.

▶—ADDBUF—◀

### INDEXAREA Option

With the INDEXAREA option you improve the input/output speed of a DIRECT INPUT or DIRECT UPDATE file with indexed data set organization, by having the highest level of index placed in main storage.

▶—INDEXAREA—(*—index-area-size—*)—◀

*index-area-size* enables you to limit the amount of main storage allowed for an index area. The size you specify must be an integer or a variable with attributes FIXED BINARY(31,0) STATIC from 0 to 64,000 in value. If you do not specify *index-area-size*, the highest level index is moved unconditionally into main storage. If you do specify *index-area-size*, the highest level index is held in main storage, provided that its size does not exceed that specified. If you specify a size less than 0 or greater than 64,000, unpredictable results will occur.

### INDEXED Option

Use the INDEXED option to define a file with indexed organization (which is described above). It is usually used with a data set created and accessed by the Indexed Sequential Access Method (ISAM), but you can also use it in some cases with VSAM data sets (as described in Chapter 11, “Defining and Using VSAM Data Sets”).

▶—INDEXED—◀

### KEYLOC Option — Key Location

Use the KEYLOC option with indexed data sets when you create the data set to specify the starting position of an embedded key in a record.

▶—KEYLOC—◀

The position, *n*, must be within the limits:

$$1 \leq n \leq \text{recordsize} - \text{keylength} + 1$$

That is, the key cannot be larger than the record, and must be contained completely within the record.

If the keys are embedded within the records, either specify the KEYLOC option, or include the DCB subparameter RKP in the DD statement for the associated data set.

If you do not specify KEYLOC, the value specified with RKP is used. If you specify neither, then RKP=0 is the default.

The KEYLOC option specifies the absolute position of an embedded key from the start of the data in a record, while the RKP subparameter specifies the position of an embedded key relative to the start of the record.

Thus the equivalent KEYLOC and RKP values for a particular byte are affected by the following:

- The KEYLOC byte count starts at 1; the RKP count starts at 0.
- The record format.

For example, if the embedded key begins at the tenth byte of a record variable, the specifications are:

```
Fixed-length:  KEYLOC(10)
                RKP=9
Variable-length: KEYLOC(10)
                RKP=13
```

If KEYLOC is specified with a value equal to or greater than 1, embedded keys exist in the record variable and on the data set. If KEYLOC is equal to zero, or is not specified, the RKP value is used. When RKP is specified, the key is part of the variable only when  $RKP \geq 1$ . As a result, embedded keys might not always be present in the record variable or the data set. If you specify KEYLOC(1), you must specify it for every file that accesses the data set. This is necessary because KEYLOC(1) cannot be converted to an unambiguous RKP value. (Its equivalent is RKP=0 for fixed format, which in turn implies nonembedded keys.) The effect of the use of both options is shown in Table 26.

*Table 26. Effect of KEYLOC and RKP Values on Establishing Embedded Keys in Record Variables or Data Sets*

KEYLOC(n)	RKP	Record variable	Data set unblocked records	Data set blocked records
n>1	RKP equivalent = $n-1+C^1$	Key	Key	Key
n=1	No equivalent	Key	Key <sup>2</sup>	Key
n=0 or not specified	RKP=C <sup>1</sup>	No Key	No Key	Key <sup>3</sup>
	RKP>C <sup>1</sup>	Key	Key	Key

**Notes:**

1. C = number of control bytes, if any:  
C=0 for fixed-length records.  
C=4 for variable-length records.
2. In this instance the key is not recognized by data management .
3. Each logical record in the block has a key.

If you specify SCALARVARYING, the embedded key must not immediately precede or follow the first byte; hence, the value specified for KEYLOC must be greater than 2.

If you include the KEYLOC option in a VSAM file declaration for checking purposes, and the key location you specify in the option conflicts with the value defined for the data set, the UNDEFINEDFILE condition is raised.

## **NOWRITE Option**

Use the NOWRITE option for DIRECT UPDATE files. It specifies that no records are to be added to the data set and that data management modules concerned solely with adding records are not required. Thus, it allows the size of the object program to be reduced.

►—NOWRITE—◄

---

## **Creating an Indexed Data Set**

When you create an indexed data set, you must open the associated file for SEQUENTIAL OUTPUT, and you must present the records in the order of ascending key values. (If there is an error in the key sequence, the KEY condition is raised.) You cannot use a DIRECT file for the creation of an indexed data set.

Table 25 on page 164 shows the statements and options for creating an indexed data set.

You can extend an indexed data set consisting of fixed-length records by adding records sequentially at the end, until the original space allocated for the prime data is filled. You must open the corresponding file for SEQUENTIAL OUTPUT and you must include DISP=MOD in the DD statement.

You can use a single DD statement to define the whole data set (index area, prime area, and overflow area), or you can use two or three statements to define the areas independently. If you use two DD statements, you can define either the index area and the prime area together, or the prime area and the overflow area together.

If you want the entire data set to be on a single volume, there is no advantage to be gained by using more than one DD statement except to define an independent overflow area (see "Overflow Area" on page 177). But, if you use separate DD statements to define the index and/or overflow area on volumes separate from that which contains the prime area, you will increase the speed of direct-access to the records in the data set by reducing the number of access mechanism movements required.

When you use two or three DD statements to define an indexed data set, the statements must appear in the order: index area; prime area; overflow area. The first DD statement must have a name (ddname), but the name fields of a second or third DD statement must be blank. The DD statements for the prime and overflow areas must specify the same type of unit (UNIT parameter). You must include all the DCB information for the data set in the first DD statement. DCB=DSORG=IS will suffice in the other statements.

## **Essential Information**

To create an indexed data set, you must give the operating system certain information either in your PL/I program or in the DD statement that defines the data set. The following paragraphs indicate the essential information, and discuss some of the optional information you can supply.

You must supply the following information when creating an indexed data set:

- Direct-access device that will write your data set (UNIT or VOLUME parameter of DD statement). Do not request DEFER.
- Block size: You can specify the block size either in your PL/I program (ENVIRONMENT attribute or LINESIZE option) or in the DD statement (BLKSIZE subparameter). If you do not specify a record length, unblocked records are the default and the record length is determined from the block size.
- Space requirements: Include space for future needs when you specify the size of the prime, index, and overflow areas. Once you have created an indexed data set, you cannot change its specification.

If you want to keep a direct-access data set (that is, you do not want the operating system to delete it at the end of your job), the DD statement must name the data set and indicate how it is to be disposed of (DSNAME and DISP parameters). The DISP parameter alone will suffice if you want to use the data set in a later step but will not need it after the end of your job.

If you want your data set stored on a particular direct-access device, you must specify the volume serial number in the DD statement (SER or REF subparameter of VOLUME parameter). If you do not specify a serial number for a data set that you want to keep, the operating system will allocate one, inform the operator, and print the number on your program listing. All the essential parameters required in a DD statement for the creation of an indexed data set are summarized in Table 27. Table 28 on page 174 lists the DCB subparameters needed. See the *MVS/370 JCL User's Guide* for a description of the DCB subparameters.

You must request space for the prime data area in the SPACE parameter. You cannot specify a secondary quantity for an indexed data set. Your request must be in units of cylinders unless you place the data set in a specific position on the volume (by specifying a track number in the SPACE parameter). In the latter case, the number of tracks you specify must be equivalent to an integral number of cylinders, and the first track must be the first track of a cylinder other than the first cylinder in the volume.

You can also use the SPACE parameter to specify the amount of space to be used for the cylinder and master indexes (unless you use a separate DD statement for this purpose). If you do not specify the space for the indexes, the operating system will use part of the independent overflow area. If there is no independent overflow area, it will use part of the prime data area.

---

*Table 27 (Page 1 of 2). Creating an Indexed Data Set: Essential Parameters of DD Statement*

<b>When required</b>	<b>What you must state</b>	<b>Parameters</b>
Always	Output device	UNIT= or VOLUME=REF=
	Storage space required	SPACE=
	Data control block information: see Table 28 on page 174	DCB=

---

Table 27 (Page 2 of 2). Creating an Indexed Data Set: Essential Parameters of DD Statement

When required	What you must state	Parameters
More than one DD statement	Name of data set and area (index, prime, overflow)	DSNAME=
Data set to be used in another job step but not required at end of job	Disposition	DISP=
Data set to be kept after end of job	Disposition Name of data set	DISP= DSNAME=
Data set to be on particular volume	Volume serial number	VOLUME=SER= or VOLUME=REF=

Table 28. DCB Subparameters for an Indexed Data Set

When required	To specify	Subparameters
These are always required <sup>2</sup>	Record format <sup>1</sup>	RECFM=F, FB, V, or VB
	Block size <sup>1</sup>	BLKSIZE=
	Data set organization	DSORG=IS
	Key length <sup>1</sup>	KEYLEN=
Include at least one of these if overflow is required	Cylinder overflow area and number of tracks per cylinder for overflow records	OPTCD=Y and CYLOFL=
	Independent overflow area	OPTCD=I
These are optional	Record length <sup>1</sup>	LRECL=
	Embedded key (relative key position) <sup>1</sup>	RKP= 2
	Master index	OPTCD=M
	Automatic processing of dummy records	OPTCD=L
	Number of data management buffers <sup>1</sup>	BUFNO=
	Number of tracks in cylinder index for each master index entry	NTM=

**Notes:**

Full DCB information must appear in the first, or only, DD statement. Subsequent statements require only DSORG=IS.

1. Or you could specify BUFNO in the ENVIRONMENT attribute.
2. RKP is required if the data set has embedded keys, unless you specify the KEYLOC option of ENVIRONMENT instead.

You must always specify the data set organization (DSORG=IS subparameter of the DCB parameter), and in the first (or only) DD statement you must also specify the length of the key (KEYLEN subparameter of the DCB parameter) unless it is specified in the ENVIRONMENT attribute.

If you want the operating system to recognize dummy records, you must code OPTCD=L in the DCB subparameter of the DD statement. This will cause the operating system to write the dummy identifier in deleted records and to ignore dummy records during sequential read processing. Do not specify OPTCD=L when using blocked or variable-length records with nonembedded keys. If you do this, the dummy record identifier (8)'1'B will overwrite the key of deleted records.

You cannot place an indexed data set on a system output (SYSOUT) device.

## Name of the Data Set

If you use only one DD statement to define your data set, you need not name the data set unless you intend to access it in another job. But if you include two or three DD statements, you must specify a data set name, even for a temporary data set.

The DSNAME parameter in a DD statement that defines an indexed data set not only gives the data set a name, but it also identifies the area of the data set to which the DD statement refers:

```
DSNAME=name(INDEX)
DSNAME=name(PRIME)
DSNAME=name(OVFLOW)
```

If you use one DD statement to define the prime and index or one DD statement to define the prime and overflow area, code DSNAME=name(PRIME). If you use one DD statement for the entire file (prime, index, and overflow), code DSNAME=name(PRIME) or simply DSNAME=name.

## Record Format and Keys

An indexed data set can contain either fixed- or variable-length records, blocked or unblocked. You must always specify the record format, either in your PL/I program (ENVIRONMENT attribute) or in the DD statement (RECFM subparameter).

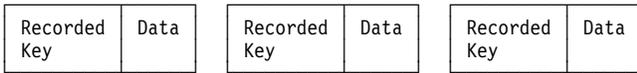
The key associated with each record can be contiguous with or embedded within the data in the record.

If the records are unblocked, the key of each record is recorded in the data set in front of the record even if it is also embedded within the record, as shown in (a) and (b) of Figure 39 on page 176.

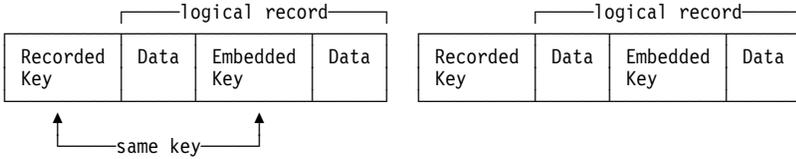
If blocked records do not have embedded keys, the key of each record is recorded within the block in front of the record, and the key of the last record in the block is also recorded just ahead of the block, as shown in (c) of Figure 39.

When blocked records have embedded keys, the individual keys are not recorded separately in front of each record in the block: the key of the last record in the block is recorded in front of the block, as shown in (d) of Figure 39.

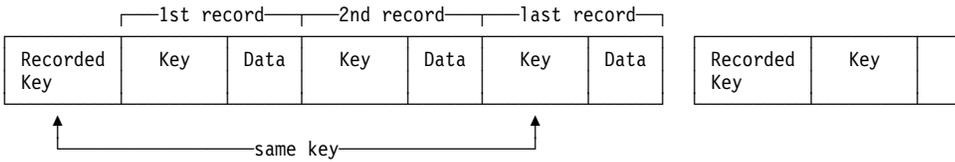
a) **Unblocked records, nonembedded keys**



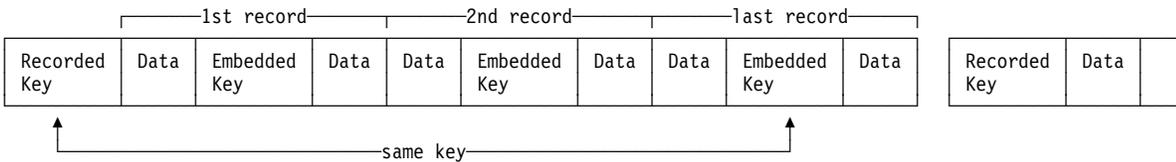
b) **Unblocked records, embedded keys**



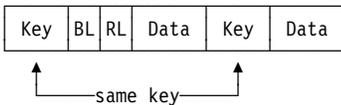
c) **Blocked records, nonembedded keys**



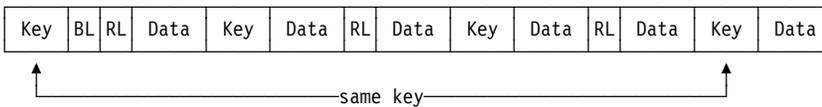
d) **Blocked records, embedded keys**



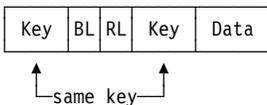
e) **Unblocked variable-length records, RKP>4**



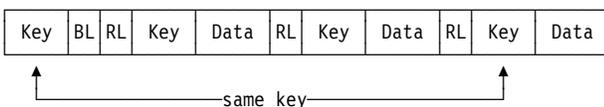
f) **Blocked variable-length records, RKP>4**



g) **Unblocked variable-length records, RKP=4**



f) **Blocked variable-length records, RKP=4**



BL = Block length  
RL = Record length

Figure 39. Record Formats in an Indexed Data Set

If you use blocked records with nonembedded keys, the record size that you specify must include the length of the key, and the block size must be a multiple of this combined length. Otherwise, record length and block size refer only to the data in the record. Record format information is shown in Figure 40.

If you use records with embedded keys, you must include the DCB subparameter RKP to indicate the position of the key within the record. For fixed-length records the value specified in the RKP subparameter is 1 less than the byte number of the first character of the key. That is, if RKP=1, the key starts in the second byte of the record. The default value if you omit this subparameter is RKP=0, which specifies that the key is not embedded in the record but is separate from it.

For variable-length records, the value you specify in the RKP subparameter must be the relative position of the key within the record plus 4. The extra 4 bytes take into account the 4-byte control field used with variable-length records. For this reason, you must never specify RKP less than 4. When deleting records, you must always specify RKP equal to or greater than 5, since the first byte of the data is used to indicate deletion.

For unblocked records, the key, even if embedded, is always recorded in a position preceding the actual data. Consequently, you do not need to specify the RKP subparameter for unblocked records.

---

RECORDS	RKP	LRECL	BLKSIZE
Blocked	Not zero	R	R * B
	Zero or omitted	R + K	B*(R+K)
Unblocked	Not zero	R	R
	Zero or omitted	R	R

R = Size of data in record  
K = Length of keys (as specified in KEYLEN subparameter)  
B = Blocking factor

Example: For blocked records, nonembedded keys, 100 bytes of data per record, 10 records per block, key length = 20:  
LRECL=120,BLKSIZE=1200,RECFM=FB

---

Figure 40. Record Format Information for an Indexed Data Set

## Overflow Area

If you intend to add records to the data set on a future occasion, you must request either a cylinder overflow area or an independent overflow area, or both.

For a cylinder overflow area, include the DCB subparameter OPTCD=Y and use the subparameter CYLOFL to specify the number of tracks in each cylinder to be reserved for overflow records. A cylinder overflow area has the advantage of a short search time for overflow records, but the amount of space available for overflow records is limited, and much of the space might be unused if the overflow records are not evenly distributed throughout the data set.

For an independent overflow area, use the DCB subparameter OPTCD=I to indicate that overflow records are to be placed in an area reserved for overflow records from all cylinders, and include a separate DD statement to define the overflow area. The use of an independent area has the advantage of reducing the amount of unused space for overflow records, but entails an increased search time for overflow records.

It is good practice to request cylinder overflow areas large enough to contain a reasonable number of additional records and an independent overflow area to be used as the cylinder overflow areas are filled.

If the prime data area is not filled during creation, you cannot use the unused portion for overflow records, nor for any records subsequently added during direct-access (although you can fill the unfilled portion of the last track used). You can reserve space for later use within the prime data area by writing dummy records during creation (see "Dummy Records" on page 167).

## Master Index

If you want the operating system to create a master index for you, include the DCB subparameter OPTCD=M, and indicate in the NTM subparameter the number of tracks in the cylinder index you wish to be referred to by each entry in the master index. The operating system will create up to three levels of master index, the first two levels addressing tracks in the next lower level of the master index.

The creation of a simple indexed data set is illustrated in Figure 41 on page 179. The data set contains a telephone directory, using the subscribers' names as keys to the telephone numbers.

---

```

//EX8#19 JOB
//STEP1 EXEC IEL1CLG
//PLI.SYSIN DD *
  TELNOS: PROC OPTIONS(MAIN);
    DCL DIREC FILE RECORD SEQUENTIAL KEYED ENV(INDEXED),
    CARD CHAR(80),
    NAME CHAR(20) DEF CARD,
    NUMBER CHAR(3) DEF CARD POS(21),
    IOFIELD CHAR(3),
    EOF BIT(1) INIT('0'B);
    ON ENDFILE(SYSIN) EOF='1'B;
    OPEN FILE(DIREC) OUTPUT;
    GET FILE(SYSIN) EDIT(CARD)(A(80));
    DO WHILE (-EOF);
    PUT FILE(SYSPRINT) SKIP EDIT (CARD) (A);
    IOFIELD=NUMBER;
    WRITE FILE(DIREC) FROM(IOFIELD) KEYFROM(NAME);
    GET FILE(SYSIN) EDIT(CARD)(A(80));
    END;
    CLOSE FILE(DIREC);
  END TELNOS;
/*
//GO.DIREC DD DSN=HPU8.TELNO(INDEX),UNIT=SYSDA,SPACE=(CYL,1),
//          DCB=(RECFM=F,BLKSIZE=3,DSORG=IS,KEYLEN=20,OPTCD=LIY,
//          CYLOFL=2),DISP=(NEW,KEEP)
//          DD DSN=HPU8.TELNO(PRIME),UNIT=SYSDA,SPACE=(CYL,1),
//          DISP=(NEW,KEEP),DCB=DSORG=IS
//          DD DSN=HPU8.TELNO(OVFLOW),UNIT=SYSDA,SPACE=(CYL,1),
//          DISP=(NEW,KEEP),DCB=DSORG=IS
//GO.SYSIN DD *
ACTION,G.          162
BAKER,R.           152
BRAMLEY,O.H.       248
CHEESEMAN,D.       141
CORY,G.            336
ELLIOTT,D.         875
FIGGINS,S.         413
HARVEY,C.D.W.     205
HASTINGS,G.M.     391
KENDALL,J.G.      294
LANCASTER,W.R.    624
MILES,R.           233
NEWMAN,M.W.       450
PITT,W.H.          515
ROLF,D.E.          114
SHEERS,C.D.       241
SUTCLIFFE,M.      472
TAYLOR,G.C.       407
WILTON,L.W.       404
WINSTONE,E.M.     307
/*

```

---

Figure 41. Creating an Indexed Data Set

---

## Accessing and Updating an Indexed Data Set

Once you create an indexed data set, you can open the file that accesses it for SEQUENTIAL INPUT or UPDATE, or for DIRECT INPUT or UPDATE. In the case of F-format records, you can also open it for OUTPUT to add records at the end of the data set. The keys for these records must have higher values than the existing keys for that data set and must be in ascending order. Table 25 on page 164 shows the statements and options for accessing an indexed data set.

Sequential input allows you to read the records in ascending key sequence, and in sequential update you can read and rewrite each record in turn. Using direct input,

you can read records using the READ statement, and in direct update you can read or delete existing records or add new ones. Sequential and direct-access are discussed in further detail below.

## Using Sequential Access

You can open a sequential file that is used to access an indexed data set with either the INPUT or the UPDATE attribute. You do not need to include source keys in the data transmission statements, nor do you need to give the file the KEYED attribute. Sequential access is in order of ascending recorded-key values. Records are retrieved in this order, and not necessarily in the order in which they were added to the data set. Dummy records are not retrieved if you include the subparameter OPTCD=L in the DD statement that defines the data set.

Except that you cannot use the EVENT option, rules governing the relationship between the READ and REWRITE statements for a SEQUENTIAL UPDATE file that accesses an indexed data set are identical to those for a consecutive data set (described in Chapter 8, “Defining and Using Consecutive Data Sets” on page 129).

You must not alter embedded keys in a record to be updated. The modified record must always overwrite the update record in the data set.

Additionally, records can be effectively deleted from the data set. Using a DELETE statement marks a record as a dummy by putting (8)'1'B in the first byte. You should not use the DELETE statement to process a data set with F-format blocked records and either KEYLOC=1 or RKP=0, or a data set with V- or VB-format records and either KEYLOC=1 or RKP=4. (The code (8)'1'B would overwrite the first byte of the recorded key.) Note that the EVENT option is not supported for SEQUENTIAL access of indexed data sets.

You can position INDEXED KEYED files opened for SEQUENTIAL INPUT and SEQUENTIAL UPDATE to a particular record within the data set by using either a READ KEY or a DELETE KEY operation that specifies the key of the desired record. Thereafter, successive READ statements without the KEY option access the next records in the data set sequentially. A subsequent READ statement without the KEY option causes the record with the next higher recorded key to be read (even if the keyed record has not been found).

Define the length of the recorded keys in an indexed data set with the KEYLENGTH ENVIRONMENT option or the KEYLEN subparameter of the DD statement that defines the data set. If the length of a source key is greater than the specified length of the recorded keys, the source key is truncated on the right.

The effect of supplying a source key that is shorter than the recorded keys in the data set differs according to whether or not you specify the GENKEY option in the ENVIRONMENT attribute. In the absence of the GENKEY option, the source key is padded on the right with blanks to the length you specify in the KEYLENGTH option of the ENVIRONMENT attribute, and the record with this padded key is read (if such a record exists). If you specify the GENKEY option, the source key is interpreted as a generic key, and the first record with a key in the class identified by this generic key is read. (For further details, see “GENKEY Option — Key Classification” on page 118.)

## Using Direct Access

You can open a direct file that is used to access an indexed data set with either the INPUT or the UPDATE attribute. You must include source keys in all data transmission statements; the DIRECT attribute implies the KEYED attribute.

You can use a DIRECT UPDATE file to retrieve, add, delete, or replace records in an indexed data set according to the following conventions:

- Retrieval** If you include the subparameter OPTCD=L in the DD statement that defines the data set, dummy records are not made available by a READ statement (the KEY condition is raised).
- Addition** A WRITE statement that includes a unique key causes a record to be inserted into the data set. If the key is the same as the recorded key of a dummy record, the new record replaces the dummy record. If the key is the same as the recorded key of a record that is not marked as deleted, or if there is no space in the data set for the record, the KEY condition is raised.
- Deletion** The record specified by the source key in a DELETE statement is retrieved, marked as deleted, and rewritten into the data set. The effect of the DELETE statement is to insert the value (8)'1'B in the first byte of the data in a record. Deletion is possible only if you specify OPTCD=L in the DD statement that defines the data set when you create it. If the data set has F-format blocked records with RKP=0 or KEYLOC=1, or V-format records with RKP=4 or KEYLOC=1, records cannot be deleted. (The code (8)'1'B would overwrite the embedded keys.)
- Replacement** The record specified by a source key in a REWRITE statement is replaced by the new record. If the data set contains F-format blocked records, a record replaced with a REWRITE statement causes an implicit READ statement to be executed unless the previous I/O statement was a READ statement that obtained the record to be replaced. If the data set contains V-format records and the updated record has a length different from that of the record read, the whole of the remainder of the track will be removed, and can cause data to be moved to an overflow track.

### Essential Information

To access an indexed data set, you must define it in one, two, or three DD statements. The DD statements must correspond with those used when the data set is created. The following paragraphs indicate the essential information you must include in each DD statement. Table 29 on page 182 summarizes this information.

Table 29. Accessing an Indexed Data Set: Essential Parameters of DD Statement

When required	What you must state	Parameters
Always	Name of data set	DSNAME=
	Disposition of data set	DISP=
	Data control block information	DCB=
If data set not cataloged	Input device	UNIT= or VOLUME=REF=
	Volume serial number	VOLUME=SER=

If the data set is cataloged, you need supply only the following information in each DD statement:

- The name of the data set (DSNAME parameter). The operating system will locate the information that describes the data set in the system catalog and, if necessary, will request the operator to mount the volume that contains it.
- Confirmation that the data set exists (DISP parameter).

If the data set is not cataloged, you must, in addition, specify the device that will process the data set and give the serial number of the volume that contains it (UNIT and VOLUME parameters).

### Example

The program in Figure 42 on page 183 updates the data set of the previous example (Figure 41 on page 179) and prints out its new contents. The input data includes the following codes to indicate the operations required:

- A** Add a new record.
- C** Change an existing record.
- D** Delete an existing record.

```

//EX8#20 JOB
//STEP1 EXEC IEL1CLG
//PLI.SYSIN DD *
DIRUPDT: PROC OPTIONS(MAIN);
    DCL DIREC FILE RECORD KEYED ENV(INDEXED),
        NUMBER CHAR(3),NAME CHAR(20),CODE CHAR(1),ONCODE BUILTIN,
        EOF BIT(1) INIT('0'B);
    ON ENDFILE(SYSIN) EOF='1'B;
    ON KEY(DIREC) BEGIN;
        IF ONCODE=51 THEN PUT FILE(SYSPRINT) SKIP EDIT
            ('NOT FOUND:',NAME)(A(15),A);
        IF ONCODE=52 THEN PUT FILE(SYSPRINT) SKIP EDIT
            ('DUPLICATE:',NAME)(A(15),A);
    END;
    OPEN FILE(DIREC) DIRECT UPDATE;
    GET FILE(SYSIN) EDIT(NAME,NUMBER,CODE)
        (COLUMN(1),A(20),A(3),A(1));
    DO WHILE (~EOF);
    PUT FILE(SYSPRINT) SKIP EDIT (' ',NAME,'#',NUMBER,' ',CODE)
        (A(1),A(20),A(1),A(3),A(1),A(1));
    SELECT (CODE);
        WHEN('A') WRITE FILE(DIREC) FROM(NUMBER) KEYFROM(NAME);
        WHEN('C') REWRITE FILE(DIREC) FROM(NUMBER) KEY(NAME);
        WHEN('D') DELETE FILE(DIREC) KEY(NAME);
        OTHERWISE PUT FILE(SYSPRINT) SKIP
            EDIT('INVALID CODE:',NAME)(A(15),A);
    END;
    GET FILE(SYSIN) EDIT(NAME,NUMBER,CODE)
        (COLUMN(1),A(20),A(3),A(1));
    END;
    CLOSE FILE(DIREC);
    PUT FILE(SYSPRINT) PAGE;
    OPEN FILE(DIREC) SEQUENTIAL INPUT;
    EOF='0'B;
    ON ENDFILE(DIREC) EOF='1'B;
    READ FILE(DIREC) INTO(NUMBER) KEYTO(NAME);
    DO WHILE (~EOF);
    PUT FILE(SYSPRINT) SKIP EDIT(NAME,NUMBER)(A);
    READ FILE(DIREC) INTO(NUMBER) KEYTO(NAME);
    END;
    CLOSE FILE(DIREC);      END DIRUPDT;

/*
//GO.DIREC DD DSN=HPUB.TELNO(INDEX),DISP=(OLD,DELETE),
//    VOL=SER=nnnnnn,UNIT=SYSDA
//    DD DSN=HPUB.TELNO(PRIME),DISP=(OLD,DELETE),
//    VOL=SER=nnnnnn,UNIT=SYSDA
//    DD DSN=HPUB.TELNO(OVFLOW),DISP=(OLD,DELETE),
//    VOL=SER=nnnnnn,UNIT=SYSDA
//GO.SYSIN DD *
NEWMAN,M.W.      516C
GOODFELLOW,D.T.  889A
MILES,R.        D
HARVEY,C.D.W.   209A
BARTLETT,S.G.   183A
CORY,G.         D
READ,K.M.       001A
PITT,W.H.
ROLF,D.E.       D
ELLIOTT,D.      291C
HASTINS,G.M.    D
BRAMLEY,O.H.    439
/*

```

Figure 42. Updating an Indexed Data Set

---

## Reorganizing an Indexed Data Set

It is necessary to reorganize an indexed data set periodically because the addition of records to the data set results in an increasing number of records in the overflow area. Therefore, even if the overflow area does not eventually become full, the average time required for the direct retrieval of a record will increase. The frequency of reorganization depends on how often you update the data set, on how much storage is available in the data set, and on your timing requirements.

Reorganizing the data set also eliminates records that are marked as “deleted” but are still present within the data set.

There are two ways to reorganize an indexed data set:

- Read the data set into an area of main storage or onto a temporary consecutive data set, and then recreate it in the original area of auxiliary storage.
- Read the data set sequentially and write it into a new area of auxiliary storage. You can then release the original auxiliary storage.

---

## Chapter 10. Defining and Using Regional Data Sets

This chapter covers regional data set organization, data transmission statements, and ENVIRONMENT options that define regional data sets. How to create and access regional data sets for each type of regional organization is then discussed.

A data set with regional organization is divided into regions, each of which is identified by a region number, and each of which can contain one record or more than one record, depending on the type of regional organization. The regions are numbered in succession, beginning with zero, and a record can be accessed by specifying its region number, and perhaps a key, in a data transmission statement.

Regional data sets are confined to direct-access devices.

Regional organization of a data set allows you to control the physical placement of records in the data set, and to optimize the access time for a particular application. Such optimization is not available with consecutive or indexed organization, in which successive records are written either in strict physical sequence or in logical sequence depending on ascending key values; neither of these methods takes full advantage of the characteristics of direct-access storage devices.

You can create a regional data set in a manner similar to a consecutive or indexed data set, presenting records in the order of ascending region numbers; alternatively, you can use direct-access, in which you present records in random sequence and insert them directly into preformatted regions. Once you create a regional data set, you can access it by using a file with the attributes SEQUENTIAL or DIRECT as well as INPUT or UPDATE. You do not need to specify either a region number or a key if the data set is associated with a SEQUENTIAL INPUT or SEQUENTIAL UPDATE file. When the file has the DIRECT attribute, you can retrieve, add, delete, and replace records at random.

Records within a regional data set are either actual records containing valid data or dummy records. The nature of the dummy records depends on the type of regional organization; the three types of regional organization are described below.

The major advantage of regional organization over other types of data set organization is that it allows you to control the relative placement of records; by judicious programming, you can optimize record access in terms of device capabilities and the requirements of particular applications.

Direct access of regional data sets is quicker than that of indexed data sets, but regional data sets have the disadvantage that sequential processing can present records in random sequence; the order of sequential retrieval is not necessarily that in which the records were presented, nor need it be related to the relative key values.

Table 30 on page 186 lists the data transmission statements and options that you can use to create and access a regional data set.

Table 30 (Page 1 of 2). Statements and options allowed for creating and accessing regional data sets

<b>File declaration<sup>1</sup></b>	<b>Valid statements,<sup>2</sup> with options you must include</b>	<b>Other options you can also include</b>
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);  LOCATE based-variable FROM(file-reference) KEYFROM(expression);	SET(pointer-reference)
SEQUENTIAL OUTPUT UNBUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	EVENT(event-reference)
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference);  READ FILE(file-reference) SET(pointer-reference);  READ FILE(file-reference) IGNORE(expression);	KEYTO(reference)  KEYTO(reference)
SEQUENTIAL INPUT UNBUFFERED	READ FILE(file-reference) INTO(reference);  READ FILE(file-reference) IGNORE(expression);	EVENT(event-reference) and/or KEYTO(reference)  EVENT(event-reference)
SEQUENTIAL UPDATE <sup>3</sup> BUFFERED	READ FILE(file-reference) INTO(reference);  READ FILE(file-reference) SET(pointer-reference);  READ FILE(file-reference) IGNORE(expression);  REWRITE FILE(file-reference);	KEYTO(reference)  KEYTO(reference)  FROM(reference)
SEQUENTIAL UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference);  READ FILE(file-reference) IGNORE(expression);  REWRITE FILE(file-reference) FROM(reference);	EVENT(event-reference) and/or KEYTO(reference)  EVENT(event-reference)  EVENT(event-reference)
DIRECT OUTPUT	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	EVENT(event-reference)
DIRECT INPUT	READ FILE(file-reference) INTO(reference) KEY(expression);	EVENT(event-reference)

Table 30 (Page 2 of 2). Statements and options allowed for creating and accessing regional data sets

File declaration <sup>1</sup>	Valid statements, <sup>2</sup> with options you must include	Other options you can also include
DIRECT UPDATE	READ FILE(file-reference) INTO(reference) KEY(expression);	EVENT(event-reference)
	REWRITE FILE(file-reference) FROM(reference) KEY(expression);	EVENT(event-reference)
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	EVENT(event-reference)
	DELETE FILE(file-reference) KEY(expression);	EVENT(event-reference)
DIRECT UPDATE EXCLUSIVE	READ FILE(file-reference) INTO(reference) KEY(expression);	EVENT(event-reference) and/or NOLOCK
	REWRITE FILE(file-reference) FROM(reference) KEY(expression);	EVENT(event-reference)
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	EVENT(event-reference)
	DELETE FILE(file-reference) KEY(expression);	EVENT(event-reference)
	UNLOCK FILE(file-reference) KEY(expression);	

**Notes:**

1. The complete file declaration would include the attributes FILE, RECORD, and ENVIRONMENT; if you use any of the options KEY, KEYFROM, or KEYTO, you must also include the attribute KEYED.
2. The statement READ FILE(file-reference); is equivalent to the statement READ FILE(file-reference) IGNORE(1);
3. The file must not have the UPDATE attribute when creating new data sets.

Regional(1) files are supported under VM with the following restrictions:

- More than one regional file with keys cannot be open at the same time.
- You must not increment KEY(TRACKID/REGION NUMBER) unless 255 records are written on the first logical track, and 256 records on each subsequent logical track.
- You must not write files with a dependency on the physical track length of a direct access device.
- When you create a file, you must specify the XTENT option of the FILEDEF command and it must be equal to the number of records in the file to be created.

The examples in this chapter are given using JCL. However, the information presented in the JCL examples is applicable to the FILEDEF VM command you

issue. For more information on the FILEDEF command, see the *VM/ESA CMS Command Reference* and the *VM/ESA CMS User's Guide*.

---

## Defining Files for a Regional Data Set

Use a file declaration with the following attributes to define a sequential regional data set:

```
DCL filename FILE RECORD
      INPUT | OUTPUT | UPDATE
      SEQUENTIAL
      BUFFERED | UNBUFFERED
      [KEYED]
      ENVIRONMENT(options);
```

To define a direct regional data set, use a file declaration with the following attributes:

```
DCL filename FILE RECORD
      INPUT | OUTPUT | UPDATE
      DIRECT
      UNBUFFERED
      [EXCLUSIVE] (cannot be used with INPUT or OUTPUT)
      ENVIRONMENT(options);
```

Default file attributes are shown in Table 15 on page 111. The file attributes are described in the *PL/I for MVS & VM Language Reference*. Options of the ENVIRONMENT attribute are discussed below.

## Specifying ENVIRONMENT Options

The ENVIRONMENT options applicable to regional data sets are:

```
REGIONAL({1|2|3})
F|V|VS|U
RECSIZE(record-length)
BLKSIZE(block-size)
SCALARVARYING
COBOL
BUFFERS(n)
KEYLENGTH(n)
NCP(n)
TRKOFL
```

### REGIONAL Option

Use the REGIONAL option to define a file with regional organization.

▶ REGIONAL ( ( 1 ) ) ◀



#### 1 | 2 | 3

specifies REGIONAL(1), REGIONAL(2), or REGIONAL(3), respectively.

#### REGIONAL(1)

specifies that the data set contains F-format records that do not have recorded keys. Each region in the data set contains only one record; therefore, each

region number corresponds to a relative record within the data set (that is, region numbers start with 0 at the beginning of the data set).

Although REGIONAL(1) data sets have no recorded keys, you can use REGIONAL(1) DIRECT INPUT or UPDATE files to process data sets that do have recorded keys. In particular, to access REGIONAL(2) and REGIONAL(3) data sets, use a file declared with REGIONAL(1) organization.

### **REGIONAL(2)**

specifies that the data set contains F-format records that have recorded keys. Each region in the data set contains only one record.

REGIONAL(2) differs from REGIONAL(1) in that REGIONAL(2) records contain recorded keys and that records are not necessarily in the specified region; the specified region identifies a starting point.

For files you create sequentially, the record is written in the specified region.

For files with the DIRECT attribute, a record is written in the first vacant space on or after the track that contains the region number you specify in the WRITE statement. For retrieval, the region number specified in the source key is employed to locate the specified region. The method of search is described further in the REGIONAL(2) discussion later in this chapter.

### **REGIONAL(3)**

specifies that the data set contains F-format, V-format, VS-format, or U-format records with recorded keys. Each region in the data set corresponds with a track on a direct-access device and can contain one or more records.

REGIONAL(3) organization is similar to REGIONAL(2) in that records contain recorded keys, but differs in that a region for REGIONAL(3) corresponds to a track and not a record position.

Direct access of a REGIONAL(3) data set employs the region number specified in a source key to locate the required region. Once the region has been located, a sequential search is made for space to add a record, or for a record that has a recorded key identical with that supplied in the source key.

VS-format records can span more than one region. With REGIONAL(3) organization, the use of VS-format removes the limitations on block size imposed by the physical characteristics of the direct-access device. If the record length exceeds the size of a track, or if there is no room left on the current track for the record, the record will be spanned over one or more tracks.

REGIONAL(1) organization is most suited to applications where there are no duplicate region numbers, and where most of the regions will be filled (reducing wasted space in the data set). REGIONAL(2) and REGIONAL(3) are more appropriate where records are identified by numbers that are thinly distributed over a wide range. You can include in your program an algorithm that derives the region number from the number that identifies a record in such a manner as to optimize the use of space within the data set; duplicate region numbers can occur but, unless they are on the same track, their only effect might be to lengthen the search time for records with duplicate region numbers.

The examples throughout this chapter illustrate typical applications of all three types of regional organization.

## Using Keys with REGIONAL Data Sets

There are two kinds of keys, recorded keys and source keys. A *recorded key* is a character string that immediately precedes each record in the data set to identify that record; its length cannot exceed 255 characters. A *source key* is the character value of the expression that appears in the KEY or KEYFROM option of a data transmission statement to identify the record to which the statement refers. When you access a record in a regional data set, the source key gives a region number, and can also give a recorded key.

You specify the length of the recorded keys in a regional data set with the KEYLENGTH option of the ENVIRONMENT attribute, or the KEYLEN subparameter on the DD statement. Unlike the keys for indexed data sets, recorded keys in a regional data set are never embedded within the record.

---

## Using REGIONAL(1) Data Sets

In a REGIONAL(1) data set, since there are no recorded keys, the region number serves as the sole identification of a particular record. The character value of the source key should represent an unsigned decimal integer that should not exceed 16777215 (although the actual number of records allowed can be smaller, depending on a combination of record size, device capacity, and limits of your access method. For direct regional(1) files with fixed format records, the maximum number of tracks which can be addressed by relative track addressing is 65,536.) If the region number exceeds this figure, it is treated as modulo 16777216; for instance, 16777226 is treated as 10. Only the characters 0 through 9 and the blank character are valid in the source key; leading blanks are interpreted as zeros. Embedded blanks are not allowed in the number; the first embedded blank, if any, terminates the region number. If more than 8 characters appear in the source key, only the rightmost 8 are used as the region number; if there are fewer than 8 characters, blanks (interpreted as zeros) are inserted on the left.

### Dummy Records

Records in a REGIONAL(1) data set are either actual records containing valid data or dummy records. A dummy record in a REGIONAL(1) data set is identified by the constant (8)'1'B in its first byte. Although such dummy records are inserted in the data set either when it is created or when a record is deleted, they are not ignored when the data set is read; your PL/I program must be prepared to recognize them. You can replace dummy records with valid data. Note that if you insert (8)'1'B in the first byte, the record can be lost if you copy the file onto a data set that has dummy records that are not retrieved.

## Creating a REGIONAL(1) Data Set

You can create a REGIONAL(1) data set either sequentially or by direct-access. Table 30 on page 186 shows the statements and options for creating a regional data set.

When you use a SEQUENTIAL OUTPUT file to create the data set, the opening of the file causes all tracks on the data set to be cleared, and a capacity record to be written at the beginning of each track to record the amount of space available on that track. You must present records in ascending order of region numbers; any region you omit from the sequence is filled with a dummy record. If there is an error in the sequence, or if you present a duplicate key, the KEY condition is raised. When the file is closed, any space remaining at the end of the current extent is filled with dummy records.

If you create a data set using a buffered file, and the last WRITE or LOCATE statement before the file is closed attempts to transmit a record beyond the limits of the data set, the CLOSE statement might raise the ERROR condition.

If you use a DIRECT OUTPUT file to create the data set, the whole primary extent allocated to the data set is filled with dummy records when the file is opened. You can present records in random order; if you present a duplicate, the existing record will be overwritten.

For sequential creation, the data set can have up to 15 extents, which can be on more than one volume. For direct creation, the data set can have only one extent, and can therefore reside on only one volume.

### Example

Creating a REGIONAL(1) data set is illustrated in Figure 43 on page 192. The data set is a list of telephone numbers with the names of the subscribers to whom they are allocated. The telephone numbers correspond with the region numbers in the data set, the data in each occupied region being a subscriber's name.

---

```

//EX9 JOB
//STEP1 EXEC IEL1CLG,PARM.PLI='NOP,MAR(1,72)',PARM.LKED='LIST'
//PLI.SYSIN DD *
CRR1: PROC OPTIONS(MAIN);
/* CREATING A REGIONAL(1) DATA SET - PHONE DIRECTORY */

DCL NOS FILE RECORD OUTPUT DIRECT KEYED ENV(REGIONAL(1));
DCL SYSIN FILE INPUT RECORD;
DCL SYSIN_REC BIT(1) INIT('1'B);
DCL 1 CARD,
    2 NAME CHAR(20),
    2 NUMBER CHAR( 2),
    2 CARD_1 CHAR(58);
DCL IOFIELD CHAR(20);

ON ENDFILE (SYSIN) SYSIN_REC = '0'B;
OPEN FILE(NOS);
READ FILE(SYSIN) INTO(CARD);

DO WHILE(SYSIN_REC);
    IOFIELD = NAME;
    WRITE FILE(NOS) FROM(IOFIELD) KEYFROM(NUMBER);
    PUT FILE(SYSIN) SKIP EDIT (CARD) (A);
    READ FILE(SYSIN) INTO(CARD);
END;

CLOSE FILE(NOS);
END CRR1;
/*
//GO.SYSLMOD DD DSN=&&GOSET,DISP=(OLD,DELETE)
//GO.NOS DD DSN=NOS,UNIT=SYSDA,SPACE=(20,100),
// DCB=(RECFM=F,BLKSIZE=20,DSORG=DA),DISP=(NEW,KEEP)
//GO.SYSIN DD *
ACTION,G. 12
BAKER,R. 13
BRAMLEY,O.H. 28
CHEESNAME,L. 11
CORY,G. 36
ELLIOTT,D. 85
FIGGINS,E.S. 43
HARVEY,C.D.W. 25
HASTINGS,G.M. 31
KENDALL,J.G. 24
LANCASTER,W.R. 64
MILES,R. 23
NEWMAN,M.W. 40
PITT,W.H. 55
ROLF,D.E. 14
SHEERS,C.D. 21
SURCLIFFE,M. 42
TAYLOR,G.C. 47
WILTON,L.W. 44
WINSTONE,E.M. 37
/*

```

---

Figure 43. Creating a REGIONAL(1) Data Set

## Accessing and Updating a REGIONAL(1) Data Set

Once you create a REGIONAL(1) data set, you can open the file that accesses it for SEQUENTIAL INPUT or UPDATE, or for DIRECT INPUT or UPDATE. You can open it for OUTPUT only if the existing data set is to be overwritten. Table 30 on page 186 shows the statements and options for accessing a regional data set.

## Sequential Access

To open a SEQUENTIAL file that is used to process a REGIONAL(1) data set, use either the INPUT or UPDATE attribute. You must not include the KEY option in data transmission statements, but the file can have the KEYED attribute, since you can use the KEYTO option. If the target character string referenced in the KEYTO option has more than 8 characters, the value returned (the 8-character region number) is padded on the left with blanks. If the target string has fewer than 8 characters, the value returned is truncated on the left.

Sequential access is in the order of ascending region numbers. All records are retrieved, whether dummy or actual, and you must ensure that your PL/I program recognizes dummy records.

Using sequential input with a REGIONAL(1) data set, you can read all the records in ascending region-number sequence, and in sequential update you can read and rewrite each record in turn.

The rules governing the relationship between READ and REWRITE statements for a SEQUENTIAL UPDATE file that accesses a REGIONAL(1) data set are identical to those for a consecutive data set. Consecutive data sets are discussed in detail in Chapter 8, “Defining and Using Consecutive Data Sets” on page 129.

## Direct Access

To open a DIRECT file that is used to process a REGIONAL(1) data set you can use either the INPUT or the UPDATE attribute. All data transmission statements must include source keys; the DIRECT attribute implies the KEYED attribute.

Use DIRECT UPDATE files to retrieve, add, delete, or replace records in a REGIONAL(1) data set according to the following conventions:

- |                    |   |
|--------------------|---|
| <b>Retrieval</b>   | All records, whether dummy or actual, are retrieved. Your program must recognize dummy records.                                 |
| <b>Addition</b>    | A WRITE statement substitutes a new record for the existing record (actual or dummy) in the region specified by the source key. |
| <b>Deletion</b>    | The record you specify by the source key in a DELETE statement is converted to a dummy record.                                  |
| <b>Replacement</b> | The record you specify by the source key in a REWRITE statement, whether dummy or actual, is replaced.                          |

## Example

Updating a REGIONAL(1) data set is illustrated in Figure 44 on page 194. Like the program in Figure 42 on page 183, this program updates the data set and lists its contents. Before each new or updated record is written, the existing record in the region is tested to ensure that it is a dummy; this is necessary because a WRITE statement can overwrite an existing record in a REGIONAL(1) data set even if it is not a dummy. Similarly, during the sequential reading and printing of the contents of the data set, each record is tested and dummy records are not printed.

```

//EX10 JOB
//STEP2 EXEC IELICLG,PARM.PLI='NOP,MAR(1,72)',PARM.LKED='LIST'
//PLI.SYSIN DD *
ACR1: PROC OPTIONS(MAIN);
/* UPDATING A REGIONAL(1) DATA SET - PHONE DIRECTORY */
DCL NOS FILE RECORD KEYED ENV(REGIONAL(1));
DCL SYSIN FILE INPUT RECORD;
DCL (SYSIN_REC,NOS_REC) BIT(1) INIT('1'B);
DCL 1 CARD,
      2 NAME CHAR(20),
      2 (NEWNO,OLDNO) CHAR( 2),
      2 CARD_1 CHAR( 1),
      2 CODE CHAR( 1),
      2 CARD_2 CHAR(54);
DCL IOFIELD CHAR(20);
DCL BYTE CHAR(1) DEF IOFIELD;

ON ENDFILE(SYSIN) SYSIN_REC = '0'B;
OPEN FILE (NOS) DIRECT UPDATE;
READ FILE(SYSIN) INTO(CARD);

DO WHILE(SYSIN_REC);
  SELECT(CODE);
  WHEN('A','C') DO;
    IF CODE = 'C' THEN
      DELETE FILE(NOS) KEY(OLDNO);
      READ FILE(NOS) KEY(NEWNO) INTO(IOFIELD);
      IF UNSPEC(BYTE) = (8)'1'B
        THEN WRITE FILE(NOS) KEYFROM(NEWNO) FROM(NAME);
      ELSE PUT FILE(SYSPRINT) SKIP LIST ('DUPLICATE:',NAME);
    END;
  WHEN('D') DELETE FILE(NOS) KEY(OLDNO);
  OTHERWISE PUT FILE(SYSPRINT) SKIP LIST ('INVALID CODE:',NAME);
  END;
  READ FILE(SYSIN) INTO(CARD);
END;

CLOSE FILE(SYSIN),FILE(NOS);
PUT FILE(SYSPRINT) PAGE;
OPEN FILE(NOS) SEQUENTIAL INPUT;
ON ENDFILE(NOS) NOS_REC = '0'B;
READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
DO WHILE(NOS_REC);
  IF UNSPEC(BYTE) ^= (8)'1'B
    THEN PUT FILE(SYSPRINT) SKIP EDIT (NEWNO,IOFIELD)(A(2),X(3),A);
  PUT FILE(SYSPRINT) SKIP EDIT (IOFIELD) (A);
  READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
  END;
CLOSE FILE(NOS);
END ACR1;
/*
//GO.NOS DD DSN=J44PLI.NOS,DISP=(OLD,DELETE),UNIT=SYSDA,VOL=SER=nnnnnn
//GO.SYSIN DD *
NEWMAN,M.W.      5640 C
GOODFELLOW,D.T.  89  A
MILES,R.        23  D
HARVEY,C.D.W.   29  A
BARTLETT,S.G.   13  A
CORY,G.         36  D
READ,K.M.       01  A
PITT,W.H.       55
ROLF,D.F.       14  D
ELLIOTT,D.     4285 C
HASTINGS,G.M.   31  D
BRAMLEY,O.H.   4928 C
*/

```

Figure 44. Updating a REGIONAL(1) Data Set

---

## Using REGIONAL(2) Data Sets

In a REGIONAL(2) data set, each record is identified by a recorded key that immediately precedes the record. The actual position of the record in the data set relative to other records is determined not by its recorded key, but by the region number that you supply in the source key of the WRITE statement that adds the record to the data set.

When you add a record to the data set by direct-access, it is written with its recorded key in the first available space after the beginning of the track that contains the region specified. When a record is read by direct-access, the search for a record with the appropriate recorded key begins at the start of the track that contains the region specified. Unless it is limited by the LIMCT subparameter of the DD statement that defines the data set, the search for a record or for space to add a record continues right through to the end of the data set and then from the beginning until the entire data set has been covered. The closer a record is to the specified region, the more quickly it can be accessed.

## Using Keys for REGIONAL(2) and (3) Data Sets

The character value of the source key can be thought of as having two logical parts—the region number and a comparison key. On output, the comparison key is written as the recorded key; for input, it is compared with the recorded key.

The rightmost 8 characters of the source key make up the region number, which must be the character representation of a fixed decimal integer that does not exceed 16777215 (although the actual number of records allowed can be smaller, depending on a combination of record size, device capacity, and limits of your access method). If the region number exceeds this figure, it is treated as modulo 16777216; for instance, 16777226 is treated as 10. You can only specify the characters 0 through 9 and the blank character; leading blanks are interpreted as zeros. Embedded blanks are not allowed in the number; the first embedded blank, if any, terminates the region number. The comparison key is a character string that occupies the left hand side of the source key, and can overlap or be distinct from the region number, from which it can be separated by other nonsignificant characters.

Specify the length of the comparison key either with the KEYLEN subparameter of the DD statement for the data set or the KEYLENGTH option of the ENVIRONMENT attribute. If the source key is shorter than the key length you specify, it is extended on the right with blanks. To retrieve a record, the comparison key must exactly match the recorded key of the record. The comparison key can include the region number, in which case the source key and the comparison key are identical; or, you can use only part of the source key. The length of the comparison key is always equal to KEYLENGTH or KEYLEN; if the source key is longer than KEYLEN+8, the characters in the source key between the comparison key and the region number are ignored.

When generating the key, you should consider the rules for conversion from arithmetic to character string. For example, the following group is incorrect:

```
DCL KEYS CHAR(8);
DO I=1 TO 10;
  KEYS=I;
  WRITE FILE(F) FROM (R)
    KEYFROM (KEYS);
END;
```

The default for I is FIXED BINARY(15,0), which requires not 8 but 9 characters to contain the character string representation of the arithmetic values. In this example the rightmost digit is truncated.

Consider the following examples of source keys (the character “b” represents a blank):

```
KEY ('JOHNbDOEbbbbbb12363251')
```

The rightmost 8 characters make up the region specification, the relative number of the record. Assume that the associated DD statement has the subparameter KEYLEN=14. In retrieving a record, the search begins with the beginning of the track that contains the region number 12363251, until the record is found having the recorded key of JOHNbDOEbbbbbb.

If the subparameter is KEYLEN=22, the search still begins at the same place, but since the comparison and the source key are the same length, the search would be for a record having the recorded key 'JOHNbDOEbbbbbb12363251'.

```
KEY ('JOHNbDOEbbbbbbDIVISIONb423bbb34627')
```

In this example, the rightmost 8 characters contain leading blanks, which are interpreted as zeros. The search begins at region number 00034627. If KEYLEN=14 is specified, the characters DIVISIONb423b will be ignored.

Assume that COUNTER is declared FIXED BINARY(21) and NAME is declared CHARACTER(15). You could specify the key like so:

```
KEY (NAME || COUNTER)
```

The value of COUNTER will be converted to a character string of 11 characters. (The rules for conversion specify that a binary value of this length, when converted to character, will result in a string of length 11—three blanks followed by eight decimal digits.) The value of the rightmost eight characters of the converted string is taken to be the region specification. Then if the keylength specification is KEYLEN=15, the value of NAME is taken to be the comparison specification.

## Dummy Records

A REGIONAL(2) data set can contain dummy records. A dummy record consists of a dummy key and dummy data. A dummy key is identified by the constant (8)'1'B in its first byte. The first byte of the data contains the sequence number of the record on the track.

The program inserts dummy records either when the data set is created or when a record is deleted. The dummy records are ignored when the program reads the data set.

However, you can replace dummy records with valid data.

## Creating a REGIONAL(2) Data Set

You can create a REGIONAL(2) data set either sequentially or by direct-access. In either case, when the file associated with the data set is opened, the data set is initialized with capacity records specifying the amount of space available on each track. Table 30 on page 186 shows the statements and options for creating a regional data set.

When you use a SEQUENTIAL OUTPUT file to create the data set, you must present records in ascending order of region numbers; any region you omit from the sequence is filled with a dummy record. If you make an error in the sequence, including attempting to place more than one record in the same region, the KEY condition is raised. When the file is closed, any space remaining at the end of the current extent is filled with dummy records.

If you create a data set using a buffered file, and the last WRITE or LOCATE statement before the file is closed attempts to transmit a record beyond the limits of the data set, the CLOSE statement can raise the ERROR condition.

If you use a DIRECT OUTPUT file to create the current extent of a data set, the whole primary extent allocated to the data set is filled with dummy records when the file is opened. You can present records in random order, and no condition is raised by duplicate keys. Each record is substituted for the first dummy record on the track that contains the region specified in the source key; if there are no dummy records on the track, the record is substituted for the first dummy record encountered on a subsequent track, unless the LIMCT subparameter specifies that the search cannot reach beyond this track. (Note that it is possible to place records with identical recorded keys in the data set).

For sequential creation, the data set can have up to 15 extents, which can be on more than one volume. For direct creation, the data set can have only one extent, and can therefore reside on only one volume.

### Example

The use of REGIONAL(2) data sets is illustrated in Figure 45 on page 198, Figure 46 on page 200, and Figure 47 on page 201. The programs in these figures perform the same functions as those given for REGIONAL(3), with which they can be compared.

The programs depict a library processing scheme, in which loans of books are recorded and reminders are issued for overdue books. Two data sets, SAMPL.STOCK and SAMPL.LOANS are used. SAMPL.STOCK contains descriptions of the books in the library, and uses the 4-digit book reference numbers as recorded keys; a simple algorithm is used to derive the region numbers from the reference numbers. (It is assumed that there are about 1000 books, each with a number in the range 1000–9999.) SAMPL.LOANS contains records of books that are on loan; each record comprises two dates, the date of issue and the date of the last reminder. Each reader is identified by a 3-digit reference number, which is used as a region number in SAMPL.LOANS; the reader and book numbers are concatenated to form the recorded keys.

Figure 45 on page 198 shows the creation of the data sets SAMPL.STOCK and SAMPL.LOANS. The file LOANS, which is used to create the data set SAMPL.LOANS, is opened for direct output to format the data set; the file is closed immediately without any records being written onto the data set. Direct creation is

also used for the data set SAMPL.STOCK because, even if the input data is presented in ascending reference number order, identical region numbers might be derived from successive reference numbers.

---

```

//EX11 JOB
//STEP1 EXEC IEL1CLG,PARM.PLI='NOP',PARM.LKED='LIST'
//PLI.SYSIN DD *
%PROCESS MAR(1,72);
/* CREATING A REGIONAL(2) DATA SET - LIBRARY LOANS          */
CRR2: PROC OPTIONS(MAIN);

DCL (LOANS,STOCK) FILE RECORD KEYED ENV(REGIONAL(2));

DCL 1 BOOK,
    2 AUTHOR CHAR(25),
    2 TITLE CHAR(50),
    2 QTY FIXED DEC(3);

DCL NUMBER CHAR(4);
DCL INTER FIXED DEC(5);
DCL REGION CHAR(8);
DCL EOF BIT(1) INIT('0'B);
/* INITIALIZE (FORMAT) LOANS DATA SET                      */

OPEN FILE(LOANS) DIRECT OUTPUT;
CLOSE FILE(LOANS);

ON ENDFILE(SYSIN) EOF='1'B;
OPEN FILE(STOCK) DIRECT OUTPUT;

GET FILE(SYSIN) SKIP LIST(NUMBER,BOOK);
DO WHILE (-EOF);
INTER = (NUMBER-1000)/9; /* REGIONS 0 TO 999              */
REGION = INTER;
WRITE FILE(STOCK) FROM (BOOK) KEYFROM(NUMBER||REGION);
PUT FILE(SYSPRINT) SKIP EDIT (BOOK) (A);
GET FILE(SYSIN) SKIP LIST(NUMBER,BOOK);
END;

CLOSE FILE(STOCK);
END CRR2;
/*
//GO.LOANS DD DSN=SAMPL.LOANS,UNIT=SYSDA,SPACE=(12,1000),
//           DCB=(RECFM=F,BLKSIZE=12,KEYLEN=7),
//           DISP=(NEW,CATLG)
//GO.STOCK DD DSN=SAMPL.STOCK,UNIT=SYSDA,SPACE=(77,1050),
//           DCB=(RECFM=F,BLKSIZE=77,KEYLEN=4),
//           DISP=(NEW,CATLG)
//GO.SYSIN DD *
'1015' 'W.SHAKESPEARE' 'MUCH ADO ABOUT NOTHING' 1
'1214' 'L.CARROLL' 'THE HUNTING OF THE SNARK' 1
'3079' 'G.FLAUBERT' 'MADAME BOVARY' 1
'3083' 'V.M.HUGO' 'LES MISERABLES' 2
'3085' 'J.K.JEROME' 'THREE MEN IN A BOAT' 2
'4295' 'W.LANGLAND' 'THE BOOK CONCERNING PIERS THE PLOWMAN' 1
'5999' 'O.KHAYYAM' 'THE RUBAIYAT OF OMAR KHAYYAM' 3
'6591' 'F.RABELAIS' 'THE HEROIC DEEDS OF GARGANTUA AND PANTAGRUEL' 1
'8362' 'H.D.THOREAU' 'WALDEN, OR LIFE IN THE WOODS' 1
'9795' 'H.G.WELLS' 'THE TIME MACHINE' 3
/*

```

Figure 45. Creating a REGIONAL(2) Data Set

## Accessing and Updating a REGIONAL(2) Data Set

Once you create a REGIONAL(2) data set, you can open the file that accesses it for SEQUENTIAL INPUT or UPDATE, or for DIRECT INPUT or UPDATE. It cannot be opened for OUTPUT. Table 30 on page 186 shows the statements and options for accessing a regional data set.

## Sequential Access

To open a SEQUENTIAL file that is used to process a REGIONAL(2) data set, use either the INPUT or UPDATE attribute. The data transmission statements must not include the KEY option, but the file can have the KEYED attribute since you can use the KEYTO option. With the KEYTO option you specify that the *recorded key only* is to be assigned to the specified variable. If the character string referenced in the KEYTO option has more characters than are specified in the KEYLEN subparameter, the value returned (the recorded key) is extended on the right with blanks; if it has fewer characters than specified by KEYLEN, the value returned is truncated on the right.

Sequential access is in the physical order in which the records exist on the data set, not necessarily in the order in which they were added to the data set. The recorded keys do not affect the order of sequential access. Dummy records are not retrieved.

The rules governing the relationship between READ and REWRITE statements for a SEQUENTIAL UPDATE file that accesses a REGIONAL(2) data set are identical with those for a CONSECUTIVE data set (described above).

## Direct Access

To open a DIRECT file that is used to process a REGIONAL(2) data set, use either the INPUT or the UPDATE attribute. You must include source keys in all data transmission statements; the DIRECT attribute implies the KEYED attribute. The search for each record is commenced at the start of the track containing the region number indicated by the key.

Using direct input, you can read any record by supplying its region number and its recorded key; in direct update, you can read or delete existing records or add new ones.

- |                    |  |
|--------------------|--|
| <b>Retrieval</b>   | Dummy records are not made available by a READ statement. The KEY condition is raised if a record with the recorded key you specify is not found.  |
| <b>Addition</b>    | A WRITE statement substitutes the new record for the first dummy record on the track containing the region specified by the source key. If there are no dummy records on this track, and you allow an extended search by the LIMCT subparameter, the new record replaces the first dummy record encountered during the search. |
| <b>Deletion</b>    | The record you specify by the source key in a DELETE statement is converted to a dummy record.   |
| <b>Replacement</b> | The record you specify by the source key in a REWRITE statement must exist; a REWRITE statement cannot be used to replace a dummy record. If it does not exist, the KEY condition is raised.   |

## Example

The data set SAMPL.LOANS, described in “Example” on page 197, is updated directly in Figure 46 on page 200. Each item of input data, read from a source input, comprises a book number, a reader number, and a code to indicate whether it refers to a new issue (I), a returned book (R), or a renewal (A). The date is written in both the issue-date and reminder-date portions of a new record or an updated record.

A sequential update of the same program is shown in the program in Figure 47 on page 201. The sequential update file (LOANS) processes the records in the data set SAMPL.LOANS, and a direct input file (STOCK) obtains the book description from the data set SAMPL.STOCK for use in a reminder note. Each record from SAMPL.LOANS is tested to see whether the last reminder was issued more than a month ago; if necessary, a reminder note is issued and the current date is written in the reminder-date field of the record.

---

```

//EX12 JOB
//STEP2 EXEC IELICLG,PARM.PLI='NOP',PARM.LKED='LIST'
//PLI.SYSIN DD *
%PROCESS MAR(1,72);
DUR2: PROC OPTIONS(MAIN);
/* UPDATING A REGIONAL(2) DATA SET DIRECTLY - LIBRARY LOANS*/

DCL LOANS FILE RECORD UPDATE DIRECT KEYED ENV(REGIONAL(2));
DCL 1 RECORD,
    2 (ISSUE,REMINDER) CHAR(6);
DCL SYSIN FILE RECORD INPUT SEQUENTIAL;
DCL SYSIN_REC BIT(1) INIT('1'B) STATIC;
DCL 1 CARD,
    2 BOOK CHAR(4),
    2 CARD_1 CHAR(5),
    2 READER CHAR(3),
    2 CARD_2 CHAR(7),
    2 CODE CHAR(1),
    2 CARD_3 CHAR(1),
    2 DATE CHAR(6), /* YYMMDD */
    2 CARD_4 CHAR(53);
DCL REGION CHAR(8) INIT(' ');

ON ENDFILE(SYSIN) SYSIN_REC = '0'B;
OPEN FILE(SYSIN), FILE(LOANS);
READ FILE(SYSIN) INTO(CARD);

DO WHILE(SYSIN_REC);
SUBSTR(REGION,6) = CARD.READER;
ISSUE,REMINDER = CARD.DATE;
PUT FILE(SYSPRINT) SKIP EDIT (CARD) (A);
SELECT(CODE);
    WHEN('I') WRITE FILE(LOANS) FROM(RECORD) /* NEW ISSUE */
              KEYFROM(READER||BOOK||REGION);
    WHEN('R') DELETE FILE(LOANS) /* RETURNED */
              KEY (READER||BOOK||REGION);
    WHEN('A') REWRITE FILE(LOANS) FROM(RECORD) /* RENEWAL */
              KEY (READER||BOOK||REGION);
    OTHERWISE PUT FILE(SYSPRINT) SKIP LIST /* INVALID CODE */
              ('INVALID CODE:',BOOK,READER);
END;
READ FILE(SYSIN) INTO(CARD);
END;

CLOSE FILE(SYSIN),FILE(LOANS);
END DUR2;
/*
//GO.SYSLMOD DD DSN=&&GOSET,DISP=(OLD,DELETE)
//GO.LOANS DD DSN=SAMPL.LOANS,DISP=(OLD,KEEP)
//GO.SYSIN DD *
5999 003 I 781221
3083 091 I 790104
1214 049 I 790205
5999 003 A 790212
3083 091 R 790212
3517 095 X 790213
*/

```

---

Figure 46. Updating a REGIONAL(2) Data Set Directly

```

//EX13 JOB
//STEP3 EXEC IEL1CLG,PARM.PLI='NOP',PARM.LKED='LIST',PARM.GO='/790308'
//PLI.SYSIN DD *
%PROCESS MAR(1,72);
SUR2: PROC OPTIONS(MAIN);
/* UPDATING A REGIONAL(2) DATA SET SEQUENTIALLY - LIBRARY LOANS */

DCL LOANS FILE RECORD SEQUENTIAL UPDATE KEYED ENV(REGIONAL(2));
DCL LOANS_REC BIT(1) INIT('1'B) STATIC;
DCL 1 RECORD,
    2 (ISSUE,REMINDER) CHAR(6);
DCL LOANKEY CHAR(7),
    READER CHAR(3) DEF LOANKEY,
    BKNO CHAR(4) DEF LOANKEY POS(4);
DCL STOCK FILE RECORD DIRECT INPUT KEYED ENV(REGIONAL(2));
DCL 1 BOOK,
    2 AUTHOR CHAR(25),
    2 TITLE CHAR(50),
    2 QTY FIXED DEC(3);
DCL TODAY CHAR(6); /* YY/MM/DD */
DCL INTER FIXED DEC(5);
DCL REGION CHAR(8);

TODAY = '790210';
OPEN FILE(LOANS),
    FILE(STOCK);
ON ENDFILE(LOANS) LOANS_REC = '0'B;
READ FILE(LOANS) INTO(RECORD) KEYTO(LOANKEY);
X = 1;

DO WHILE(LOANS_REC);
    PUT FILE(SYSPRINT) SKIP EDIT
    (X,'REM DATE ',REMINDER,' TODAY ',TODAY) (A(3),A(9),A,A(7),A);
    X = X+1;

    IF REMINDER < TODAY THEN /* ? LAST REMINDER ISSUED */
        DO; /* MORE THAN A MONTH AGO*/
            INTER = (BKNO-1000)/9; /* YES, PRINT NEW REMINDER*/
            REGION = INTER;
            READ FILE(STOCK) INTO(BOOK) KEY(BKNO||REGION);
            REMINDER = TODAY; /* UPDATE REMINDER DATE */
            PUT FILE(SYSPRINT) SKIP EDIT
            ('NEW REM DATE',REMINDER,READER,AUTHOR,TITLE)
            (A(12),A,X(2),A,X(2),A,X(2),A);
            REWRITE FILE(LOANS) FROM(RECORD);
            END;
        READ FILE(LOANS) INTO(RECORD) KEYTO(LOANKEY);
    END;

CLOSE FILE(LOANS),FILE(STOCK);
END SUR2;
/*
//GO.SYSLMOD DD DSN=&&G0SET,DISP=(OLD,DELETE)
//GO.LOANS DD DSN=SAMPL.LOANS,DISP=(OLD,KEEP)
//GO.STOCK DD DSN=SAMPL.STOCK,DISP=(OLD,KEEP)
/*

```

Figure 47. Updating a REGIONAL(2) Data Set Sequentially

---

## Using REGIONAL(3) Data Sets

A REGIONAL(3) data set differs from a REGIONAL(2) data set (described above) only in the following respects:

- Each region number identifies a *track* on the direct-access device that contains the data set; the region number should not exceed 32767. A region in excess of 32767 is treated as modulo 32768; for example, 32778 is treated as 10.
- A region can contain one or more records, or a segment of a VS-format record.
- The data set can contain F-format, V-format, VS-format, or U-format records. You can create dummy records, but a data set that has V-format, VS-format, or U-format records is not preformatted with dummy records because the lengths of records cannot be known until they are written; however, all tracks in the primary extent are cleared and the operating system maintains a *capacity record* at the beginning of each track, in which it records the amount of space available on that track.

Source keys for a REGIONAL(3) data set are interpreted exactly as those for a REGIONAL(2) data set are, and the search for a record or space to add a record is conducted in a similar manner.

### Dummy Records

Dummy records for REGIONAL(3) data sets with F-format records are identical to those for REGIONAL(2) data sets.

You can identify V-format, VS-format, and U-format dummy records because they have dummy recorded keys ((8)'1'B in the first byte). The four control bytes in each V-format and VS-format dummy record are retained, but the contents of V-format, VS-format, and U-format dummy records are undefined. V-format, VS-format, and U-format records convert to dummy records only when a record is deleted, and you cannot reconvert them to valid records.

## Creating a REGIONAL(3) Data Set

You can create a REGIONAL(3) data set either sequentially or by direct-access. In either case, when the file associated with the data set is opened, the data set is initialized with capacity records specifying the amount of space available on each track. Table 30 on page 186 shows the statements and options for creating a regional data set.

When you use a SEQUENTIAL OUTPUT file to create the data set, you must present records in ascending order of region numbers, but you can specify the same region number for successive records. For F-format records, any record you omit from the sequence is filled with a dummy record. If you make an error in the sequence, the KEY condition is raised. If a track becomes filled by records for which the same region number was specified, the region number is incremented by one; an attempt to add a further record with the same region number raises the KEY condition (sequence error).

If you create a data set using a buffered file, and the last WRITE or LOCATE statement before the file is closed attempts to transmit a record beyond the limits of the data set, the CLOSE statement can raise the ERROR condition.

If you use a DIRECT OUTPUT file to create the data set, the whole primary extent allocated to the data set is initialized when the data set is opened. For F-format records, the space is filled with dummy records, and for V-format, VS-format, and U-format records, the capacity record for each track is written to indicate empty tracks. You can present records in random order, and no condition is raised by duplicate keys or duplicate region specifications. If the data set has F-format records, each record is substituted for the first dummy record in the region (track) specified on the source key; if there are no dummy records on the track, and you allow an extended search by the LIMCT subparameter, the record is substituted for the first dummy record encountered during the search. If the data set has V-format, VS-format, or U-format records, the new record is inserted on the specified track, if sufficient space is available; otherwise, if you allow an extended search, the new record is inserted in the next available space.

Note that for spanned records, space might be required for overflow onto subsequent tracks.

For sequential creation, the data set can have up to 15 extents, which can be on more than one volume. For direct creation, the data set can have only one extent, and can therefore reside on only one volume.

### Example

A program for creating a REGIONAL(3) data set is shown in Figure 48. This program is similar to creating a REGIONAL(2) data set, discussed in “Example” on page 197 and illustrated in Figure 45 on page 198. The only important difference is that in REGIONAL(3) the data set SAMPL.STOCK is created sequentially. In REGIONAL(3) data sets, duplicate region numbers are acceptable, because each region can contain more than one record.

---

```
//EX14 JOB
//STEP1 EXEC IEL1CLG,PARM.PLI='NOP',PARM.LKED='LIST'
//PLI.SYSIN DD *
%PROCESS MAR(1,72);
/* CREATING A REGIONAL(3) DATA SET - LIBRARY LOANS */
CRR3: PROC OPTIONS(MAIN);

DCL LOANS FILE RECORD KEYED ENV(REGIONAL(3));

DCL STOCK FILE RECORD KEYED ENV(REGIONAL(3));
DCL 1 BOOK,
    2 AUTHOR CHAR(25),
    2 TITLE CHAR(50),
    2 QTY FIXED DEC(3);

DCL NUMBER CHAR(4);
DCL INTER FIXED DEC(5);
DCL REGION CHAR(8);
DCL EOF BIT(1) INIT('0'B);
```

---

Figure 48 (Part 1 of 2). Creating a REGIONAL(3) Data Set

---

```

/* INITIALIZE (FORMAT) LOANS DATA SET                               */
OPEN FILE(LOANS) DIRECT OUTPUT;
CLOSE FILE(LOANS);

ON ENDFILE(SYSIN) EOF='1'B;
OPEN FILE(STOCK) SEQUENTIAL OUTPUT;

GET FILE(SYSIN) SKIP LIST(NUMBER,BOOK);
DO WHILE (~EOF);
INTER = (NUMBER-1000)/2250; /* REGIONS = 0,1,2,3,4 FOR A DEVICE */
                               /* HOLDING 200 (OR MORE) BOOKS/TRACK*/
REGION = INTER;
WRITE FILE(STOCK) FROM(BOOK) KEYFROM(NUMBER||REGION);
PUT FILE(SYSPRINT) SKIP EDIT (BOOK) (A);
GET FILE(SYSIN) SKIP LIST(NUMBER,BOOK);
END;

CLOSE FILE(STOCK);
END CRR3;
/*
//GO.LOANS DD DSN=SAMPL.LOANS,UNIT=SYSDA,SPACE=(TRK,3),
//          DCB=(RECFM=F,BLKSIZE=12,KEYLEN=7),
//          DISP=(NEW,CATLG)
//GO.STOCK DD DSN=SAMPL.STOCK,UNIT=SYSDA,SPACE=(TRK,5),
//          DCB=(RECFM=F,BLKSIZE=77,KEYLEN=4),
//          DISP=(NEW,CATLG)
//GO.SYSIN DD *
'1015' 'W.SHAKESPEARE' 'MUCH ADO ABOUT NOTHING' 1
'1214' 'L.CARROLL' 'THE HUNTING OF THE SNARK' 1
'3079' 'G.FLAUBERT' 'MADAME BOVARY' 1
'3083' 'V.M.HUGO' 'LES MISERABLES' 2
'3085' 'J.K.JEROME' 'THREE MEN IN A BOAT' 2
'4295' 'W.LANGLAND' 'THE BOOK CONCERNING PIERS THE PLOWMAN' 1
'5999' 'O.KHAYYAM' 'THE RUBAIYAT OF OMAR KHAYYAM' 3
'6591' 'F.RABELAIS' 'THE HEROIC DEEDS OF GARGANTUA AND PANTAGRUEL' 1
'8362' 'H.D.THOREAU' 'WALDEN, OR LIFE IN THE WOODS' 1
'9795' 'H.G.WELLS' 'THE TIME MACHINE' 3
/*

```

---

Figure 48 (Part 2 of 2). Creating a REGIONAL(3) Data Set

## Accessing and Updating a REGIONAL(3) Data Set

Once you create a REGIONAL(3) data set, you can open the file that accesses it for SEQUENTIAL INPUT or UPDATE, or for DIRECT INPUT or UPDATE. You can only open it for OUTPUT if the entire existing data set is to be deleted and replaced. Table 30 on page 186 shows the statements and options for accessing a regional data set.

### Sequential Access

To open a SEQUENTIAL file that is used to access a REGIONAL(3) data set, use either the INPUT or UPDATE attribute. You must not include the KEY option in the data transmission statements, but the file can have the KEYED attribute since you can use the KEYTO option.

With the KEYTO option you can specify that the *recorded key only* is to be assigned to the specified variable. If the character string referenced in the KEYTO option has more characters than you specify in the KEYLEN subparameter, the value returned (the recorded key) is extended on the right with blanks; if it has fewer characters than you specify by KEYLEN, the value returned is truncated on the right.

Sequential access is in the order of ascending relative tracks. Records are retrieved in this order, and not necessarily in the order in which they were added to the data set. The recorded keys do not affect the order of sequential access. Dummy records are not retrieved.

The rules governing the relationship between READ and REWRITE statements for a SEQUENTIAL UPDATE file that accesses a REGIONAL(3) data set are identical with those for a CONSECUTIVE data set (described above).

### Direct Access

To open a DIRECT file that is used to process a REGIONAL(3) data set, use either the INPUT or the UPDATE attribute. You must include source keys in all data transmission statements; the DIRECT attribute implies the KEYED attribute.

Using direct input, you can read any record by supplying its region number and its recorded key; in direct update, you can read or delete existing records or add new ones.

**Retrieval** Dummy records are not made available by a READ statement. The KEY condition is raised if a record with the specified recorded key is not found.

**Addition** In a data set with F-format records, a WRITE statement substitutes the new record for a dummy record in the region (track) specified by the source key. If there are no dummy records on the specified track, and you use the LIMCT subparameter to allow an extended search, the new record replaces the first dummy record encountered during the search. If the data set has V-format, VS-format, or U-format records, a WRITE statement inserts the new record after any records already present on the specified track if space is available; otherwise, if you allow an extended search, the new record is inserted in the next available space.

**Deletion** A record you specify by the source key in a DELETE statement is converted to a dummy record. You can re-use the space formerly occupied by an F-format record; space formerly occupied by V-format, VS-format, or U-format records is not available for reuse.

**Replacement** The record you specify by the source key in a REWRITE statement must exist; you cannot use a REWRITE statement to replace a dummy record. When a VS-format record is replaced, the new one must not be shorter than the old.

**Note:** If a track contains records with duplicate recorded keys, the record farthest from the beginning of the track will never be retrieved during direct-access.

### Example

Updating REGIONAL(3) data sets is shown in the following two figures, Figure 49 on page 206 and Figure 50 on page 207. These are similar to the REGIONAL(2) figures, Figure 46 on page 200 and Figure 47 on page 201.

You should note that REGIONAL(3) updating differs from REGIONAL(2) updating in only one important way. When you update the data set directly, illustrated in Figure 49 on page 206, the region number for the data set SAMPL.LOANS is obtained simply by testing the reader number.

Sequential updating, shown in Figure 50 on page 207, is very much like Figure 47 on page 201, the REGIONAL(2) example.

---

```

//EX15 JOB
//STEP2 EXEC IEL1CLG,PARM.PLI='NOP',PARM.LKED='LIST'
//PLI.SYSIN DD *
%PROCESS MAR(1,72);
DUR3: PROC OPTIONS(MAIN);
/* UPDATING A REGIONAL(3) DATA SET DIRECTLY - LIBRARY LOANS */
DCL LOANS FILE RECORD UPDATE DIRECT KEYED ENV(REGIONAL(3));
DCL 1 RECORD,
    2 (ISSUE,REMINDER) CHAR(6);
DCL SYSIN FILE RECORD INPUT SEQUENTIAL;
DCL SYSIN_REC BIT(1) INIT('1'B);
DCL 1 CARD,
    2 BOOK CHAR(4),
    2 CARD_1 CHAR(5),
    2 READER CHAR(3),
    2 CARD_2 CHAR(7),
    2 CODE CHAR(1),
    2 CARD_3 CHAR(1),
    2 DATE CHAR(6),
    2 CARD_4 CHAR(53);
DCL REGION CHAR(8);
ON ENDFILE(SYSIN) SYSIN_REC= '0'B;
OPEN FILE(SYSIN),FILE(LOANS);
READ FILE(SYSIN) INTO(CARD);
DO WHILE(SYSIN_REC);
    ISSUE,REMINDER = DATE;
    SELECT;
        WHEN(READER < '034') REGION = '00000000';
        WHEN(READER < '067') REGION = '00000001';
        OTHERWISE REGION = '00000002';
    END;
    SELECT(CODE);
        WHEN('I') WRITE FILE(LOANS) FROM(RECORD)
            KEYFROM(READER||BOOK||REGION);
        WHEN('R') DELETE FILE(LOANS)
            KEY (READER||BOOK||REGION);
        WHEN('A') REWRITE FILE(LOANS) FROM(RECORD)
            KEY (READER||BOOK||REGION);
        OTHERWISE PUT FILE(SYSPRINT) SKIP LIST
            ('INVALID CODE: ',BOOK,READER);
    END;
    PUT FILE(SYSPRINT) SKIP EDIT (CARD) (A);
    READ FILE(SYSIN) INTO(CARD);
    END;
CLOSE FILE(SYSIN),FILE(LOANS);
END DUR3;
/*
//GO.SYSLMOD DD DSN=&&GOSSET,DISP=(OLD,DELETE)
//GO.LOANS DD DSN=SAMPL.LOANS,DISP=(OLD,KEEP)
//GO.SYSIN DD *
5999 003 I 781221
3083 091 I 790104
1214 049 I 790205
5999 003 A 790212
3083 091 R 790212
3517 095 X 790213
*/

```

---

Figure 49. Updating a REGIONAL(3) Data Set Directly

---

```

//EX16 JOB
//STEP3 EXEC IEL1CLG,PARM.PLI='NOP',PARM.LKED='LIST',PARM.GO='/790308'
//PLI.SYSIN DD *
%PROCESS MAR(1,72);
SUR3: PROC OPTIONS(MAIN);
/* UPDATING A REGIONAL(3) DATA SET SEQUENTIALLY - LIBRARY LOANS */

DCL LOANS FILE RECORD SEQUENTIAL UPDATE KEYED ENV(REGIONAL(3));
DCL LOANS_REC BIT(1) INIT('1'B);
DCL 1 RECORD,
    2 (ISSUE,REMINDER) CHAR(6);
DCL LOANKEY CHAR(7),
    READER CHAR(3) DEF LOANKEY,
    BKNO CHAR(4) DEF LOANKEY POS(4);
DCL STOCK FILE RECORD DIRECT INPUT KEYED ENV(REGIONAL(3));
DCL 1 BOOK,
    2 AUTHOR CHAR(25),
    2 TITLE CHAR(50),
    2 QTY FIXED DEC(3);
DCL TODAY CHAR(6);/*YYMMDD*/
DCL INTER FIXED DEC(5),
    REGION CHAR(8);

TODAY = '790210';
OPEN FILE (LOANS), FILE(STOCK);
ON ENDFILE(LOANS) LOANS_REC = '0'B;
READ FILE(LOANS) INTO(RECORD) KEYTO(LOANKEY);
X = 1;

DO WHILE(LOANS_REC);
PUT FILE(SYSPRINT) SKIP EDIT
(X,'REM DATE ',REMINDER,' TODAY ',TODAY) (A(3),A(9),A,A(7),A);
X = X+1;

IF REMINDER < TODAY THEN
DO;
INTER = (BKNO-1000)/2250;
REGION = INTER;
READ FILE(STOCK) INTO(BOOK) KEY(BKNO||REGION);
REMINDER = TODAY;
PUT FILE(SYSPRINT) SKIP EDIT
('NEW REM DATE',REMINDER,READER,AUTHOR,TITLE)
(A(12),A,X(2),A,X(2),A,X(2),A);
REWRITE FILE(LOANS) FROM(RECORD);
END;
READ FILE(LOANS) INTO(RECORD) KEYTO(LOANKEY);
END;

CLOSE FILE(LOANS),FILE(STOCK);
END SUR3;
/*
//GO.LOANS DD DSN=SAMPL.LOANS,DISP=(OLD,KEEP)
//GO.STOCK DD DSN=SAMPL.STOCK,DISP=(OLD,KEEP)

```

---

Figure 50. Updating a REGIONAL(3) Data Set Sequentially

---

## Essential Information for Creating and Accessing Regional Data Sets

To create a regional data set, you must give the operating system certain information, either in your PL/I program or in the DD statement that defines the data set. The following paragraphs indicate the essential information, and discuss some of the optional information you can supply.

You must supply the following information when creating a regional data set:

- Device that will write your data set (UNIT or VOLUME parameter of DD statement).
- Block size: You can specify the block size either in your PL/I program (in the BLKSIZE option of the ENVIRONMENT attribute) or in the DD statement (BLKSIZE subparameter). If you do not specify a record length, unblocked records are the default and the record length is determined from the block size.

If you want to keep a data set (that is, you do not want the operating system to delete it at the end of your job), the DD statement must name the data set and indicate how it is to be disposed of (DSNAME and DISP parameters). The DISP parameter alone will suffice if you want to use the data set in a later step but do not need it after the end of your job.

If you want your data set stored on a particular direct-access device, you must indicate the volume serial number in the DD statement (SER or REF subparameter of VOLUME parameter). If you do not supply a serial number for a data set that you want to keep, the operating system allocates one, informs the operator, and prints the number on your program listing. All the essential parameters required in a DD statement for the creation of a regional data set are summarized in Table 31 on page 209; and Table 32 on page 210 lists the DCB subparameters needed. See your *MVS/ESA JCL User's Guide* for a description of the DCB subparameters.

You cannot place a regional data set on a system output (SYSOUT) device.

In the DCB parameter, you must always specify the data set organization as direct by coding DSORG=DA. You cannot specify the DUMMY or DSN=NULLFILE parameters in a DD statement for a regional data set. For REGIONAL(2) and REGIONAL(3), you must also specify the length of the recorded key (KEYLEN) unless it is specified in the ENVIRONMENT attribute; see "Using Keys for REGIONAL(2) and (3) Data Sets" on page 195 for a description of how the recorded key is derived from the source key supplied in the KEYFROM option.

For REGIONAL(2) and REGIONAL(3), if you want to restrict the search for space to add a new record, or the search for an existing record, to a limited number of tracks beyond the track that contains the specified region, use the LIMCT subparameter of the DCB parameter. If you omit this parameter, the search continues to the end of the data set, and then from the beginning of the data set back to the starting point.

Table 31. Creating a regional data set: essential parameters of the DD statement

When required	What you must state	Parameters
Always	Output device <sup>1</sup>	UNIT= or VOLUME=REF=
	Storage space required <sup>2</sup>	SPACE=
	Data control block information: see Table 32 on page 210	DCB=
Data set to be used in another job step but not required in another job	Disposition	DISP=
Data set to be kept after end of job	Disposition	DISP=
	Name of data set	DSNAME=
Data set to be on particular volume	Volume serial number	VOLUME=SER= or VOLUME=REF=

<sup>1</sup>Regional data sets are confined to direct-access devices.

<sup>2</sup>For sequential access, the data set can have up to 15 extents, which can be on more than one volume. For creation with DIRECT access, the data set can have only one extent.

To access a regional data set, you must identify it to the operating system in a DD statement. The following paragraphs indicate the minimum information you must include in the DD statement; this information is summarized in Table 33 on page 210.

If the data set is cataloged, you only need to supply the following information in your DD statement:

- The name of the data set (DSNAME parameter). The operating system locates the information that describes the data set in the system catalog and, if necessary, requests the operator to mount the volume that contains it.
- Confirmation that the data set exists (DISP parameter).

If the data set is not cataloged, you must, in addition, specify the device that will read the data set and give the serial number of the volume that contains the data set (UNIT and VOLUME parameters).

Unlike indexed data sets, regional data sets do not require the subparameter OPTCD=L in the DD statement.

When opening a multiple-volume regional data set for sequential update, the ENDFILE condition is raised at the end of the first volume.

*Table 32. DCB subparameters for a regional data set*

<b>When required</b>	<b>To specify</b>	<b>Subparameters</b>
These are always required	Record format <sup>1</sup>	RECFM=F or RECFM=V <sup>2</sup> REGIONAL(3) only, or RECFM=U REGIONAL(3) only
	Block size <sup>1</sup>	BLKSIZE=
	Data set organization	DSORG=DA
	Key length (REGIONAL(2) and (3) only) <sup>1</sup>	KEYLEN=
These are optional	Limited search for a record or space to add a record (REGIONAL(2) and (3) only)	LIMCT=
	Number of data management buffers <sup>1</sup>	BUFNO=

<sup>1</sup>Or you can specify the block size in the ENVIRONMENT attribute.

<sup>2</sup>RECFM=VS must be specified in the ENVIRONMENT attribute for sequential input or update.

*Table 33. Accessing a regional data set: essential parameters of the DD statement*

<b>When required</b>	<b>What you must state</b>	<b>Parameters</b>
Always	Name of data set	DSNAME=
	Disposition of data set	DISP=
If data set not cataloged	Input device	UNIT= or VOLUME=REF=
	Volume serial number	VOLUME=SER=

---

## Chapter 11. Defining and Using VSAM Data Sets

This chapter covers VSAM (the Virtual Storage Access Method) organization for record-oriented data transmission, VSAM ENVIRONMENT options, compatibility with other PL/I data set organizations, and the statements you use to load and access the three types of VSAM data sets that PL/I supports—entry-sequenced, key-sequenced, and relative record. The chapter is concluded by a series of examples showing the PL/I statements, Access Method Services commands, and JCL statements necessary to create and access VSAM data sets.

For additional information about the facilities of VSAM, the structure of VSAM data sets and indexes, the way in which they are defined by Access Method Services, and the required JCL statements, see the VSAM publications for your system.

PL/I supports the use of VSAM data sets under VM. VSAM under VM has some restrictions. See the *VM/ESA CMS User's Guide* for those restrictions.

---

### Using VSAM Data Sets

#### How to Run a Program with VSAM Data Sets

Before you execute a program that accesses a VSAM data set, you need to know:

- The name of the VSAM data set
- The name of the PL/I file
- Whether you intend to share the data set with other users

Then you can write the required DD statement to access the data set:

```
//filename DD DSNAME=dsname,DISP=OLD|SHR
```

For example, if your file is named PL1FILE, your data set named VSAMDS, and you want exclusive control of the data set, enter:

```
//PL1FILE DD DSNAME=VSAMDS,DISP=OLD
```

To share your data set, use DISP=SHR.

For a PL/I program originally written for ISAM data sets that requires a simulation of ISAM data-set handling, you need to use the AMP parameter of the DD statement. You might also want to use it to optimize VSAM's performance.

To optimize VSAM's performance by controlling the number of VSAM buffers used for your data set, see the VSAM publications.

#### Pairing an Alternate Index Path with a File

When using an alternate index, you simply specify the name of the *path* in the DSNAME parameter of the DD statement associating the base data set/alternate index pair with your PL/I file. Before using an alternate index, you should be aware of the restrictions on processing; these are summarized in Table 35 on page 218.

Given a PL/I file called PL1FILE and the alternate index path called PERSALPH, the DD statement required would be:

```
//PL1FILE DD DSN=PERSALPH,DISP=OLD
```

---

## VSAM Organization

PL/I provides support for three types of VSAM data sets:

- Key-sequenced data sets (KSDS)
- Entry-sequenced data sets (ESDS)
- Relative record data sets (RRDS).

These correspond roughly to PL/I indexed, consecutive, and regional data set organizations, respectively. They are all ordered, and they can all have keys associated with their records. Both sequential and keyed access are possible with all three types.

Although only key-sequenced data sets have keys as part of their logical records, keyed access is also possible for entry-sequenced data sets (using relative-byte addresses) and relative record data sets (using relative record numbers).

All VSAM data sets are held on direct-access storage devices, and a virtual storage operating system is required to use them.

The physical organization of VSAM data sets differs from those used by other access methods. VSAM does not use the concept of blocking, and, except for relative record data sets, records need not be of a fixed length. In data sets with VSAM organization, the data items are arranged in *control intervals*, which are in turn arranged in *control areas*. For processing purposes, the data items within a control interval are arranged in logical records. A control interval can contain one or more logical records, and a logical record can span two or more control intervals. Concern about blocking factors and record length is largely removed by VSAM, although records cannot exceed the maximum specified size. VSAM allows access to the control intervals, but this type of access is not supported by PL/I.

VSAM data sets can have two types of indexes—prime and alternate. A *prime index* is the index to a KSDS that is established when you define a data set; it always exists and can be the only index for a KSDS. You can have one or more *alternate indexes* on a KSDS or an ESDS. Defining an alternate index for an ESDS enables you to treat the ESDS, in general, as a KSDS. An alternate index on a KSDS enables a field in the logical record different from that in the prime index to be used as the key field. Alternate indexes can be either *nonunique*, in which duplicate keys are allowed, or *unique*, in which they are not. The prime index can never have duplicate keys.

Any change in a data set that has alternate indexes must be reflected in all the indexes if they are to remain useful. This activity is known as *index upgrade*, and is done by VSAM for any index in the *index upgrade set* of the data set. (For a KSDS, the prime index is always a member of the index upgrade set.) However, you must avoid making changes in the data set that would cause duplicate keys in the prime index or in a unique alternate index.

Before using a VSAM data set for the first time, you need to define it to the system with the DEFINE command of Access Method Services, which you can use to completely define the type, structure, and required space of the data set. This command also defines the data set's indexes (together with their key lengths and locations) and the index upgrade set if the data set is a KSDS or has one or more alternate indexes. A VSAM data set is thus "created" by Access Method Services.

The operation of writing the initial data into a newly created VSAM data set is referred to as *loading* in this publication.

Use the three different types of data sets according to the following purposes:

- Use *entry-sequenced data sets* for data that you primarily access in the order in which it was created (or the reverse order).
- Use *key-sequenced data sets* when you normally access records through keys within the records (for example, a stock-control file where the part number is used to access a record).
- Use *relative record data sets* for data in which each item has a particular number, and you normally access the relevant record by that number (for example, a telephone system with a record associated with each number).

You can access records in all types of VSAM data sets either directly by means of a key, or sequentially (backward or forward). You can also use a combination of the two ways: Select a starting point with a key and then read forward or backward from that point.

You can create alternate indexes for key-sequenced and entry-sequenced data sets. You can then access your data in many sequences or by one of many keys. For example, you could take a data set held or indexed in order of employee number and index it by name in an alternate index. Then you could access it in alphabetic order, in reverse alphabetic order, or directly using the name as a key. You could also access it in the same kind of combinations by employee number.

Figure 51 on page 214 and Table 34 on page 215 show how the same data could be held in the three different types of VSAM data sets and illustrates their respective advantages and disadvantages.

The diagrams show how the information contained in the family tree below could be held in VSAM data sets of different types.

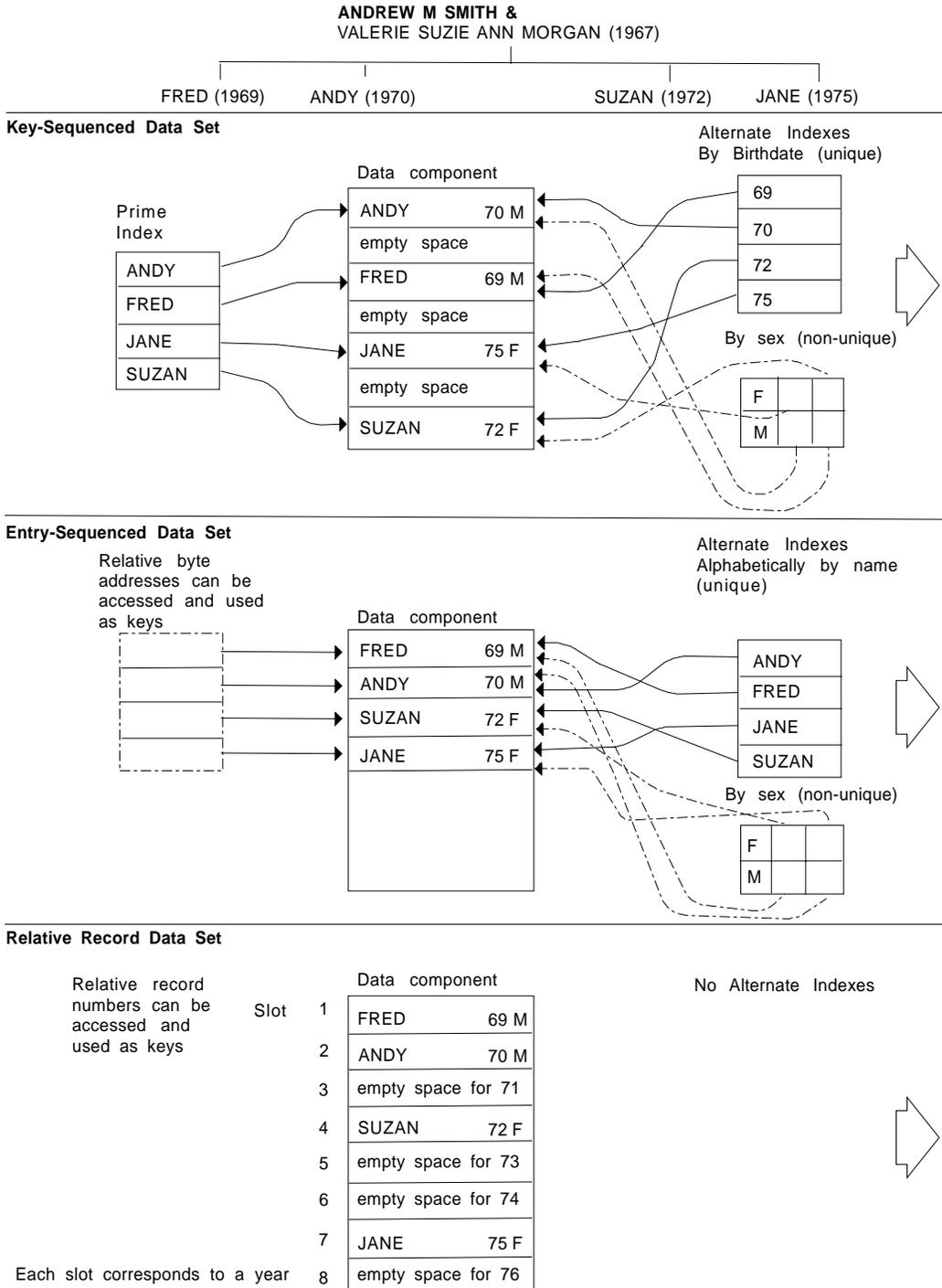


Figure 51. Information Storage in VSAM Data Sets of Different Types

Table 34. Types and Advantages of VSAM Data Sets

Data set type	Method of loading	Method of reading	Method of updating	Pros and cons
Key-Sequenced	Sequentially in order or prime index which must be unique	KEYED by specifying key of record in prime or unique alternate index  SEQUENTIAL backward or forward in order of any index  Positioning by key followed by sequential reading either backward or forward	KEYED specifying a unique key in any index  SEQUENTIAL following positioning by unique key  Record deletion allowed  Record insertion allowed	<b>Advantages</b> Complete access and updating  <b>Disadvantages</b> Records must be in order of prime index before loading  <b>Uses</b> For uses where access will be related to key
Entry-Sequenced	Sequentially (forward only)  The RBA of each record can be obtained and used as a key	SEQUENTIAL backward or forward  KEYED using unique alternate index or RBA  Positioning by key followed by sequential either backward or forward	New records at end only  Existing records cannot have length changed  Access can be sequential or KEYED using alternate index  Record deletion not allowed	<b>Advantages</b> Simple fast creation  No requirement for a unique index  <b>Disadvantages</b> Limited updating facilities  <b>Uses</b> For uses where data will primarily be accessed sequentially
Relative Record	Sequentially starting from slot 1  KEYED specifying number of slot  Positioning by key followed by sequential writes	KEYED specifying numbers as key  Sequential forward or backward omitting empty records	Sequentially starting at a specified slot and continuing with next slot  Keyed specifying numbers as key  Record deletion allowed  Record insertion into empty slots allowed	<b>Advantages</b> Speedy access to record by number  <b>Disadvantages</b> Structure tied to numbering sequences  No alternate index Fixed length records  <b>Uses</b> For use where records will be accessed by number

## Keys for VSAM Data Sets

All VSAM data sets can have keys associated with their records. For key-sequenced data sets, and for entry-sequenced data sets accessed via an alternate index, the key is a defined field within the logical record. For entry-sequenced data sets, the key is the *relative byte address* (RBA) of the record. For relative-record data sets, the key is a *relative record number*.

## Keys for Indexed VSAM Data Sets

Keys for key-sequenced data sets and for entry-sequenced data sets accessed via an alternate index are part of the logical records recorded on the data set. You define the length and location of the keys when you create the data set.

The ways you can reference the keys in the KEY, KEYFROM, and KEYTO options are as described under “KEY(expression) Option,” “KEYFROM(expression) Option,” and “KEYTO(reference) Option” in Chapter 12 of the *PL/I for MVS & VM Language Reference*. See also “Using keys” on page 163.

## Relative Byte Addresses (RBA)

Relative byte addresses allow you to use keyed access on an ESDS associated with a KEYED SEQUENTIAL file. The RBAs, or keys, are character strings of length 4, and their values are defined by VSAM. You cannot construct or manipulate RBAs in PL/I; you can, however, compare their values in order to determine the relative positions of records within the data set. RBAs are not normally printable.

You can obtain the RBA for a record by using the KEYTO option, either on a WRITE statement when you are loading or extending the data set, or on a READ statement when the data set is being read. You can subsequently use an RBA obtained in either of these ways in the KEY option of a READ or REWRITE statement.

Do not use an RBA in the KEYFROM option of a WRITE statement.

VSAM allows use of the relative byte address as a key to a KSDS, but this use is not supported by PL/I.

## Relative Record Numbers

Records in an RRDS are identified by a relative record number that starts at 1 and is incremented by 1 for each succeeding record. You can use these relative record numbers as keys for keyed access to the data set.

Keys used as relative record numbers are character strings of length 8. The character value of a source key you use in the KEY or KEYFROM option must represent an unsigned integer. If the source key is not 8 characters long, it is truncated or padded with blanks (interpreted as zeros) on the *left*. The value returned by the KEYTO option is a character string of length 8, with leading zeros suppressed.

## Choosing a Data Set Type

When planning your program, the first decision to be made is which type of data set to use. There are three types of VSAM data sets and five types of non-VSAM data sets available to you. VSAM data sets can provide all the function of the other types of data sets, plus additional function available only in VSAM. VSAM can usually match other data set types in performance, and often improve upon it. However, VSAM is more subject to performance degradation through misuse of function.

The comparison of all eight types of data sets given in Table 16 on page 122 is helpful; however, many factors in the choice of data set type for a large installation are beyond the scope of this book.

Figure 51 on page 214 shows you the possibilities available with the types of VSAM data sets. When choosing between the VSAM data set types, you should base your choice on the most common sequence in which you will require your data. The following is a suggested procedure that you can use to help ensure a combination of data sets and indexes that provide the function you require.

1. Determine the type of data and how it will be accessed.
  - a. Primarily sequentially — favors ESDS.
  - b. Primarily by key — favors KSDS.
  - c. Primarily by number — favors RRDS.
2. Determine how you will load the data set. Note that you must load a KSDS in key sequence; thus an ESDS with an alternate index path can be a more practical alternative for some applications.
3. Determine whether you require access through an alternate index path. These are only supported on KSDS and ESDS. If you require an alternate index path, determine whether the alternate index will have unique or nonunique keys. Use of nonunique keys can limit key processing. However, it might also be impractical to assume that you will use unique keys for all future records; if you attempt to insert a record with a nonunique key in an index that you have created for unique keys, it will cause an error.
4. When you have determined the data sets and paths that you require, ensure that the operations you have in mind are supported. Figure 52 on page 218 and Table 35 on page 218 might be helpful.

Do not try to access a dummy VSAM data set, because you will receive an error message indicating that you have an undefined file.

Table 36 on page 228, Table 37 on page 232, and Table 39 on page 247 show the statements allowed for entry-sequenced data sets, indexed data sets, and relative record data sets, respectively.

	SEQUENTIAL	KEYED SEQUENTIAL	DIRECT
<b>INPUT</b>	ESDS	ESDS	KSDS
	KSDS	KSDS	RRDS
	RRDS	RRDS	Path(U)
	Path(N)	Path(N)	
	Path(U)	Path(U)	
<b>OUTPUT</b>	ESDS	ESDS	KSDS
	RRDS	KSDS	RRDS
		RRDS	Path(U)
<b>UPDATE</b>	ESDS	ESDS	KSDS
	KSDS	KSDS	RRDS
	RRDS	RRDS	Path(U)
	Path(N)	Path(N)	
	Path(U)	Path(U)	

Key: ESDS Entry-sequenced data set  
 KSDS Key-sequenced data set  
 RRDS Relative record data set  
 Path(N) Alternate index path with nonunique keys  
 Path(U) Alternate index path with unique keys

You can combine the attributes on the left with those at the top of the figure for the data sets and paths shown. For example, only an ESDS and an RRDS can be SEQUENTIAL OUTPUT.

PL/I does not support dummy VSAM data sets.

Figure 52. VSAM Data Sets and Allowed File Attributes

Table 35. Processing Allowed on Alternate Index Paths

Base cluster type	Alternate index key type	Processing	Restrictions
KSDS	Unique key	As normal KSDS	Cannot modify key of access.
	Nonunique key	Limited keyed access	Cannot modify key of access.
ESDS	Unique key	As KSDS	No deletion. Cannot modify key of access.
	Nonunique key	Limited keyed access	No deletion. Cannot modify key of access.

## Defining Files for VSAM Data Sets

You define a sequential VSAM data set by using a file declaration with the following attributes:

```
DCL filename FILE RECORD
      INPUT | OUTPUT | UPDATE
      SEQUENTIAL
      BUFFERED
      [KEYED]
      ENVIRONMENT(options);
```

You define a direct VSAM data set by using a file declaration with the following attributes:

```
DCL filename FILE RECORD
      INPUT | OUTPUT | UPDATE
      DIRECT
      UNBUFFERED
      [KEYED]
      ENVIRONMENT(options);
```

Table 15 on page 111 shows the default attributes. The file attributes are described in the *PL/I for MVS & VM Language Reference*. Options of the ENVIRONMENT attribute are discussed below.

Some combinations of the file attributes INPUT or OUTPUT or UPDATE and DIRECT or SEQUENTIAL or KEYED SEQUENTIAL are allowed only for certain types of VSAM data sets. Figure 52 on page 218 shows the compatible combinations.

## Specifying ENVIRONMENT Options

Many of the options of the ENVIRONMENT attribute affecting data set structure are not needed for VSAM data sets. If you specify them, they are either ignored or are used for checking purposes. If those that are checked conflict with the values defined for the data set, the UNDEFINEDFILE condition is raised when an attempt is made to open the file.

The ENVIRONMENT options applicable to VSAM data sets are:

```
BKWD
BUFND(n)
BUFNI(n)
BUFSP(n)
COBOL
GENKEY
PASSWORD(password-specification)
REUSE
SCALARVARYING
SIS
SKIP
VSAM
```

COBOL, GENKEY, and SCALARVARYING options have the same effect as they do when you use them for non-VSAM data sets.

The options that are checked for a VSAM data set are RECSIZE and, for a key-sequenced data set, KEYLENGTH and KEYLOC. NCP has meaning when you are using the ISAM compatibility interface. Table 15 on page 111 shows which options are ignored for VSAM. Table 15 on page 111 also shows the required and default options.

For VSAM data sets, you specify the maximum and average lengths of the records to the Access Method Services utility when you define the data set. If you include the RECSIZE option in the file declaration for checking purposes, specify the maximum record size. If you specify RECSIZE and it conflicts with the values defined for the data set, the UNDEFINEDFILE condition is raised.

## BKWD Option

Use the BKWD option to specify backward processing for a SEQUENTIAL INPUT or SEQUENTIAL UPDATE file associated with a VSAM data set.

▶—BKWD—◀

Sequential reads (that is, reads without the KEY option) retrieve the previous record in sequence. For indexed data sets, the previous record is, in general, the record with the next lower key. However, if you are accessing the data set via a nonunique alternate index, records with the same key are recovered in their normal sequence. For example, if the records are:

A B C1 C2 C3 D E

where C1, C2, and C3 have the same key, they are recovered in the sequence:

E D C1 C2 C3 B A

When a file with the BKWD option is opened, the data set is positioned at the last record. ENDFILE is raised in the normal way when the start of the data set is reached.

Do not specify the BKWD option with either the REUSE option or the GENKEY option. Also, the WRITE statement is not allowed for files declared with the BKWD option.

## BUFND Option

Use the BUFND option to specify the number of data buffers required for a VSAM data set.

▶—BUFND—(—*n*—)◀

**n** specifies an integer, or a variable with attributes FIXED BINARY(31) STATIC.

Multiple data buffers help performance when the file has the SEQUENTIAL attribute and you are processing long groups of contiguous records sequentially.

## BUFNI Option

Use the BUFNI option to specify the number of index buffers required for a VSAM key-sequenced data set.

▶—BUFNI—(—*n*—)◀

**n** specifies an integer, or a variable with the attributes FIXED BINARY(31) STATIC.

Multiple index buffers help performance when the file has the KEYED attribute. Specify at least as many index buffers as there are levels in the index.

## BUFSP Option

Use the BUFSP option to specify, in bytes, the total buffer space required for a VSAM data set (for both the data and index components).

►►—BUFSP—(*n*)—

**n** specifies an integer, or a variable with the attributes FIXED BINARY(31) STATIC.

It is usually preferable to specify the BUFNI and BUFND options rather than BUFSP.

## GENKEY Option

For the description of this option, see “GENKEY Option — Key Classification” on page 118.

## PASSWORD Option

When you define a VSAM data set to the system (using the DEFINE command of Access Method Services), you can associate READ and UPDATE passwords with it. From that point on, you must include the appropriate password in the declaration of any PL/I file that you use to access the data set.

►►—PASSWORD—(*password-specification*)—

### password-specification

is a character constant or character variable that specifies the password for the type of access your program requires. If you specify a constant, it must not contain a repetition factor; if you specify a variable, it must be level-1, element, static, and unsubscripted.

The character string is padded or truncated to 8 characters and passed to VSAM for inspection. If the password is incorrect, the system operator is given a number of chances to specify the correct password. You specify the number of chances to be allowed when you define the data set. After this number of unsuccessful tries, the UNDEFINEDFILE condition is raised.

The three levels of password supported by PL/I are:

- Master
- Update
- Read.

Specify the highest level of password needed for the type of access that your program performs.

## REUSE Option

Use the REUSE option to specify that an OUTPUT file associated with a VSAM data set is to be used as a work file.

►►—REUSE—

The data set is treated as an empty data set each time the file is opened. Any secondary allocations for the data set are released, and the data set is treated exactly as if it were being opened for the first time.

Do not associate a file that has the REUSE option with a data set that has alternate indexes or the BKWD option, and do not open it for INPUT or UPDATE.

The REUSE option takes effect only if you specify REUSE in the Access Method Services DEFINE CLUSTER command.

### **SIS Option**

The SIS option is applicable to key-sequenced data sets accessed by means of a DIRECT file.

▶—SIS—◀

If you use mass sequential insert for a VSAM data set (that is, if you insert records with ascending keys), a KEYED SEQUENTIAL UPDATE file is normally appropriate. In this case, however, VSAM delays writing the records to the data set until a complete control interval has been built. If you specify DIRECT, VSAM writes each record as soon as it is presented. Thus, in order to achieve immediate writing *and* faster access with efficient use of disk space, use a DIRECT file and specify the SIS option.

The SIS option is intended primarily for use in online applications.

It is never an error to specify (or omit) the SIS option; its effect on performance is significant only in the circumstances described.

### **SKIP Option**

Use the SKIP option of the ENVIRONMENT attribute to specify that the VSAM OPTCD "SKP" is to be used wherever possible. It is applicable to key-sequenced data sets that you access by means of a KEYED SEQUENTIAL INPUT or UPDATE file.

▶—SKIP—◀

You should specify this option for the file if your program accesses individual records scattered throughout the data set, but does so primarily in ascending key order.

Omit this option if your program reads large numbers of records sequentially without the use of the KEY option, or if it inserts large numbers of records at specific points in the data set (mass sequential insert).

It is never an error to specify (or omit) the SKIP option; its effect on performance is significant only in the circumstances described.

## VSAM Option

Specify the VSAM option for VSAM data sets, unless you also intend to use the file to access non-VSAM data sets (if this is the case, see “Using the VSAM Compatibility Interface” on page 225).

▶—VSAM—▶

## Performance Options

SKIP, SIS, BUFND, BUFNI, and BUFSP are options you can specify to optimize VSAM's performance. You can also specify the buffer options in the AMP parameter of the DD statement; they are explained in your Access Method Services manual.

---

## Defining Files for Alternate Index Paths

VSAM allows you to define alternate indexes on key sequenced and entry sequenced data sets. This enables you to access key sequenced data sets in a number of ways other than from the prime index. This also allows you to index and access entry sequenced data sets by key or sequentially in order of the keys. Consequently, data created in one form can be accessed in a large number of different ways. For example, an employee file might be indexed by personnel number, by name, and also by department number.

When an alternate index has been built, you actually access the data set through a third object known as an alternate index *path* that acts as a connection between the alternate index and the data set.

Two types of alternate indexes are allowed—unique key and nonunique key. For a unique key alternate index, each record must have a different alternate key. For a nonunique key alternate index, any number of records can have the same alternate key. In the example suggested above, the alternate index using the names could be a unique key alternate index (provided each person had a different name). The alternate index using the department number would be a nonunique key alternate index because more than one person would be in each department. An example of alternate indexes applied to a family tree is given in Figure 51 on page 214.

In most respects, you can treat a data set accessed through a unique key alternate index path like a KSDS accessed through its prime index. You can access the records by key or sequentially, you can update records, and you can add new records. If the data set is a KSDS, you can delete records, and alter the length of updated records. Restrictions and allowed processing are shown in Table 35 on page 218. When you add or delete records, all indexes associated with the data set are by default altered to reflect the new situation.

In data sets accessed through a nonunique key alternate index path, the record accessed is determined by the key and the sequence. The key can be used to establish positioning so that sequential access can follow. The use of the key accesses the first record with that key. When the data set is read backwards, only the order of the keys is reversed. The order of the records with the same key remains the same whichever way the data set is read.

---

## Using Files Defined for non-VSAM Data Sets

In most cases, if your PL/I program uses files declared with ENVIRONMENT (CONSECUTIVE) or ENVIRONMENT(INDEXED) or with no ENVIRONMENT, it can access VSAM data sets without alteration. If your program uses REGIONAL files, you must alter it and recompile before it can use VSAM data sets. PL/I can detect that a VSAM data set is being opened and can provide the correct access, either directly or by use of a compatibility interface.

If your PL/I program uses REGIONAL(1) files, it cannot be used unaltered to access VSAM relative-record data sets.

The aspects of compatibility that affect your usage of VSAM if your data sets or programs were created for other access methods are as follows:

- The recreation of your data sets as VSAM data sets. The Access Method Services REPRO command recreates data sets in VSAM format. This command is described in the *MVS/DFP Access Method Services* manual.
- All VSAM key-sequenced data sets have embedded keys, even if they have been converted from ISAM data sets with nonembedded keys.
- JCL DD statement changes.
- The unaltered use of your programs with VSAM data sets. This is described in the following section.
- The alteration of your programs to allow them to use VSAM data sets. A brief discussion of this is given later in this section.

## CONSECUTIVE Files

For CONSECUTIVE files, compatibility depends on the ability of the PL/I routines to recognize the data set type and use the correct access method.

You should realize, however, that there is no concept of fixed-length records in VSAM. Therefore, if your program relies on the RECORD condition to detect incorrect length records, it will not function in the same way using VSAM data sets as it does with non-VSAM data sets.

## INDEXED Files

Complete compatibility is provided for INDEXED files. For files that you declare with the INDEXED ENVIRONMENT option, the PL/I library routines recognize a VSAM data set and will process it as VSAM.

However, because ISAM record handling differs in detail from VSAM record handling, use of VSAM processing might not always give the required result. To ensure complete compatibility with PL/I ENV(INDEXED) files, VSAM provides the compatibility interface—a program that simulates ISAM-type handling of VSAM data sets.

Because VSAM does not support EXCLUSIVE files, your program will not be compatible on VSAM and ISAM if it relies on this feature.

## Using the VSAM Compatibility Interface

The compatibility interface simulates ISAM-type handling on VSAM key-sequenced data sets. This allows compatibility for any program whose logic depends on ISAM-type record handling. The VSAM compatibility interface is VSAM supplied (See the VSAM publications.)

The compatibility interface is used when you specify the RECFM or OPTCD keyword in a DD statement associated with a file declared with the INDEXED ENVIRONMENT option, or when you use an NCP value greater than 1 in the ENVIRONMENT option. These conditions are taken by the PL/I library routines to mean that the compatibility interface is required. Choose the RECFM value, either F, V, or VS, to match the type of record that would be used by an ISAM data set. Use the OPTCD value "OPTCD=I," which is the default, if you require complete ISAM compatibility (see 3 below).

You cannot use the compatibility interface for a data set having a nonzero RKP (KEYLOC) and RECFM=F. If your program uses such files you must recompile to change the INDEXED file declaration to VSAM.

You need the compatibility interface in the following circumstances:

1. If your program uses nonembedded keys.
2. If your program relies on the raising of the RECORD condition when an incorrect-length record is encountered.
3. If your program relies on checking for deleted records. In ISAM, deleted records remain in the data set but are flagged as deleted. In VSAM, they become inaccessible to you, and their space is available for overwriting.

*Note on Deletion:* If you want the compatibility interface but want deletion of records handled in the VSAM manner, you must use 'OPTCD=IL' in the DD statement.

An example of DD statements that would result in the compatibility interface being used when accessing a VSAM data set is:

```
//PLIFILE DD DSNAME=VSAM1,  
//          DISP=OLD,AMP='RECFM=F'
```

or, to use the compatibility interface with VSAM-type deletion of records:

```
//PLIFILE DD DSNAME=VSAM1,  
//          DISP=OLD,AMP='OPTCD=IL'
```

## Adapting Existing Programs for VSAM

You can readily adapt existing programs with indexed, consecutive, or REGIONAL(1) files for use with VSAM data sets. As indicated above, programs with consecutive files might not need alteration, and there is never any necessity to alter programs with indexed files unless you wish to avoid the use of the compatibility interface or if the logic depends on EXCLUSIVE files. Programs with REGIONAL(1) data sets require only minor revision. Programs with REGIONAL(2) or REGIONAL(3) files need restructuring before you can use them with VSAM data sets.

## **CONSECUTIVE Files**

If the logic of the program depends on raising the RECORD condition when a record of an incorrect length is found, you will have to write your own code to check for the record length and take the necessary action. This is because records of any length up to the maximum specified are allowed in VSAM data sets.

## **INDEXED Files**

You need to change programs using indexed (that is, ISAM) files only if you wish to avoid using the compatibility interface.

You should remove dependence on the RECORD condition, and insert your own code to check for record length if this is necessary.

Also remove any checking for deleted records.

## **REGIONAL(1) Files**

You can alter programs using REGIONAL(1) data sets to use VSAM relative record data sets.

Remove REGIONAL(1) and any other non-VSAM ENVIRONMENT options from the file declaration and replace them with ENV(VSAM).

Also remove any checking for deleted records, because VSAM deleted records are not accessible to you.

## **Using Several Files in One VSAM Data Set**

You can associate multiple files with one VSAM data set in the following ways:

- Use a common DD statement. You can use the TITLE option of the OPEN statement for this purpose, as described on page spotref=assfile..
- Use separate DD statements, ensure that the DD statements reference the same data set name, or a path accessing the same underlying VSAM data set. PL/I OPEN specifies the VSAM MACRF=DSN option, indicating that VSAM is to share control blocks based on a common data set name.

In both cases, PL/I creates one set of control blocks—an Access Method Control Block and a Request Parameter List (RPL)—for each file and does not provide for associating multiple RPLs with a single ACB. These control blocks are described in the VSAM publications. and normally need not concern you.

Multiple files can perform retrievals against a single data set with no difficulty. However, if one or more files perform updates, the following can occur:

- There is a risk that other files will retrieve down-level records. You can avoid this by having all files open with the UPDATE attribute.
- When more than one file is open with the UPDATE attribute, retrieval of any record in a control interval makes all other records in that control interval unavailable until the update is complete. This raises the ERROR condition with condition code 1027 if a second file tries to access one of the unavailable records. You could design your application to retry the retrieval after completion of the other file's data transmission, or you can avoid the error by not having two files associated with the same data set at one time.
- When one or more of the multiple files is an alternate index path, an update through an alternate index path might update the alternate index before the data record is written, resulting in a mismatch between index and data.

## Using Shared Data Sets

PL/I does not support cross-region or cross-system sharing of data sets.

---

## Defining VSAM Data Sets

Use the DEFINE CLUSTER command of Access Method Services to define and catalog VSAM data sets. To use the DEFINE command, you need to know:

- The name and password of the master catalog if the master catalog is password protected
- The name and password of the VSAM private catalog you are using if you are not using the master catalog
- Whether VSAM space for your data set is available
- The type of VSAM data set you are going to create
- The volume on which your data set is to be placed
- The average and maximum record size in your data set
- The position and length of the key for an indexed data set
- The space to be allocated for your data set
- How to code the DEFINE command
- How to use the Access Method Services program.

When you have the information, you are in a position to code the DEFINE command and then define and catalog the data set using Access Method Services.

## Entry-Sequenced Data Sets

The statements and options allowed for files associated with an ESDS are shown in Table 36.

Table 36 (Page 1 of 2). Statements and Options Allowed for Loading and Accessing VSAM Entry-Sequenced Data Sets

File declaration <sup>1</sup>	Valid statements, with options you must include	Other options you can also include
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference);	KEYTO(reference)
	LOCATE based-variable FILE(file-reference);	SET(pointer-reference)
SEQUENTIAL OUTPUT UNBUFFERED	WRITE FILE(file-reference) FROM(reference);	EVENT(event-reference) and/or KEYTO(reference)
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference);	KEYTO(reference) or KEY(expression) <sup>3</sup>
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference) or KEY(expression) <sup>3</sup>
	READ FILE(file-reference);	IGNORE(expression)
SEQUENTIAL INPUT UNBUFFERED	READ FILE(file-reference) INTO(reference);	EVENT(event-reference) and/or either KEY(expression) <sup>3</sup> KEYTO(reference)
	READ FILE(file-reference); <sup>2</sup>	EVENT(event-reference) and/or IGNORE(expression)
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);	KEYTO(reference) or KEY(expression) <sup>3</sup>
	READ FILE(file-reference) SET(pointer-reference);	KEYTO(reference) or KEY(expression) <sup>3</sup>
	READ FILE(file-reference) <sup>2</sup>	IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference);	KEYTO(reference)
	REWRITE FILE(file-reference);	FROM(reference) and/or KEY(expression) <sup>3</sup>

Table 36 (Page 2 of 2). Statements and Options Allowed for Loading and Accessing VSAM Entry-Sequenced Data Sets

File declaration <sup>1</sup>	Valid statements, with options you must include	Other options you can also include
SEQUENTIAL UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference);	EVENT(event-reference) and/or either KEY(expression) <sup>3</sup> or KEYTO(reference)
	READ FILE(file-reference); <sup>2</sup>	EVENT(event-reference) and/or IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference);	EVENT(event-reference) and/or KEYTO(reference)
	REWRITE FILE(file-reference) FROM(reference);	EVENT(event-reference) and/or KEY(expression) <sup>3</sup>

**Notes:**

1. The complete file declaration would include the attributes FILE, RECORD, and ENVIRONMENT; if you use either of the options KEY or KEYTO, it must also include the attribute KEYED.
2. The statement "READ FILE(file-reference);" is equivalent to the statement "READ FILE(file-reference) IGNORE (1);".
3. The expression used in the KEY option must be a relative byte address, previously obtained by means of the KEYTO option.

## Loading an ESDS

When an ESDS is being loaded, the associated file must be opened for SEQUENTIAL OUTPUT. The records are retained in the order in which they are presented.

You can use the KEYTO option to obtain the relative byte address of each record as it is written. You can subsequently use these keys to achieve keyed access to the data set.

## Using a SEQUENTIAL File to Access an ESDS

You can open a SEQUENTIAL file that is used to access an ESDS with either the INPUT or the UPDATE attribute. If you use either of the options KEY or KEYTO, the file must also have the KEYED attribute.

Sequential access is in the order that the records were originally loaded into the data set. You can use the KEYTO option on the READ statements to recover the RBAs of the records that are read. If you use the KEY option, the record that is recovered is the one with the RBA you specify. Subsequent sequential access continues from the new position in the data set.

For an UPDATE file, the WRITE statement adds a new record at the end of the data set. With a REWRITE statement, the record rewritten is the one with the specified RBA if you use the KEY option; otherwise, it is the record accessed on the previous READ. You must not attempt to change the length of the record that is being replaced with a REWRITE statement.

The DELETE statement is not allowed for entry-sequenced data sets.

## Defining and Loading an ESDS

In Figure 53 on page 231, the data set is defined with the DEFINE CLUSTER command and given the name PLIVSAM.AJC1.BASE. The NONINDEXED keyword causes an ESDS to be defined.

The PL/I program writes the data set using a SEQUENTIAL OUTPUT file and a WRITE FROM statement. The DD statement for the file contains the DSNAME of the data set given in the NAME parameter of the DEFINE CLUSTER command.

The RBA of the records could have been obtained during the writing for subsequent use as keys in a KEYED file. To do this, a suitable variable would have to be declared to hold the key and the WRITE...KEYTO statement used. For example:

```
DCL CHARS CHAR(4);  
WRITE FILE(FAMFILE) FROM (STRING)  
KEYTO(CHARS);
```

Note that the keys would not normally be printable, but could be retained for subsequent use.

The cataloged procedure IEL1CLG is used. Because the same program (in Figure 53 on page 231) can be used for adding records to the data set, it is retained in a library. This procedure is shown in the next example.

---

```

//OPT9#7 JOB
//STEP1 EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
DEFINE CLUSTER -
  (NAME(PLIVSAM.AJC1.BASE) -
  VOLUMES(nnnnnn) -
  NONINDEXED -
  RECORDSIZE(80 80) -
  TRACKS(2 2))
/*
//STEP2 EXEC IEL1CLG
//PLI.SYSIN DD *
CREATE: PROC OPTIONS(MAIN);

DCL
  FAMFILE FILE SEQUENTIAL OUTPUT ENV(VSAM),
  IN FILE RECORD INPUT,
  STRING CHAR(80),
  EOF BIT(1) INIT('0'B);

ON ENDFILE(IN) EOF='1'B;

READ FILE(IN) INTO (STRING);
DO I=1 BY 1 WHILE (~EOF);
  PUT FILE(SYSPRINT) SKIP EDIT (STRING) (A);
  WRITE FILE(FAMFILE) FROM (STRING);
  READ FILE(IN) INTO (STRING);
END;

PUT SKIP EDIT(I-1,' RECORDS PROCESSED')(A);
END;
/*
//LKED.SYSLMOD DD DSN=HPU8.MYDS(PGMA),DISP=(NEW,CATLG),
// UNIT=SYSDA,SPACE=(CYL,(1,1,1))
//GO.FAMFILE DD DSN=PLIVSAM.AJC1.BASE,DISP=OLD
//GO.IN DD *
FRED          69          M
ANDY          70          M
SUZAN        72          F
/*

```

---

Figure 53. Defining and Loading an Entry-Sequenced Data Set (ESDS)

## Updating an Entry-Sequenced Data Set

Figure 54 shows the addition of a new record on the end of an ESDS. This is done by executing again the program shown in Figure 53. A SEQUENTIAL OUTPUT file is used and the data set associated with it by use of the DSNAME parameter specifying the name PLIVSAM.AJC1.BASE specified in the DEFINE command shown in Figure 53.

---

```

//OPT9#8 JOB
//STEP1 EXEC PGM=PGMA
//STEPLIB DD DSN=HPU8.MYDS(PGMA),DISP=(OLD,KEEP)
// DD DSN=CEE.V1R2M0.SCEERUN,DISP=SHR
//SYSPRINT DD SYSOUT=A
//FAMFILE DD DSN=PLIVSAM.AJC1.BASE,DISP=SHR
//IN DD *
JANE          75          F
//

```

---

Figure 54. Updating an ESDS

You can rewrite existing records in an ESDS, provided that the length of the record is not changed. You can use a SEQUENTIAL or KEYED SEQUENTIAL update file

to do this. If you use keys, they can be the RBAs or keys of an alternate index path.

Delete is not allowed for ESDS.

## Key-Sequenced and Indexed Entry-Sequenced Data Sets

The statements and options allowed for indexed VSAM data sets are shown in Table 37. An indexed data set can be a KSDS with its prime index, or either a KSDS or an ESDS with an alternate index. Except where otherwise stated, the following description applies to all indexed VSAM data sets.

Table 37 (Page 1 of 3). Statements and Options Allowed for Loading and Accessing VSAM Indexed Data Sets

File declaration <sup>1</sup>	Valid statements, with options you must include	Other options you can also include
SEQUENTIAL OUTPUT BUFFERED <sup>3</sup>	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);  LOCATE based-variable FILE(file-reference) KEYFROM(expression);	SET(pointer-reference)
SEQUENTIAL OUTPUT UNBUFFERED <sup>3</sup>	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	EVENT(event-reference)
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference);  READ FILE(file-reference) SET(pointer-reference);  READ FILE(file-reference); <sup>2</sup>	KEY(expression) or KEYTO(reference)  KEY(expression) or KEYTO(reference)  IGNORE(expression)
SEQUENTIAL INPUT UNBUFFERED	READ FILE(file-reference) INTO(reference);  READ FILE(file-reference); <sup>2</sup>	EVENT(event-reference) and/or either KEY(expression) or KEYTO(reference)  EVENT(event-reference) and/or IGNORE(expression)
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);  READ FILE(file-reference) SET(pointer-reference);  READ FILE(file-reference); <sup>2</sup>  WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);  REWRITE FILE(file-reference);  DELETE FILE(file-reference) <sup>5</sup>	KEY(expression) or KEYTO(reference)  KEY(expression) or KEYTO(reference)  IGNORE(expression)  FROM(reference) and/or KEY(expression)  KEY(expression)

Table 37 (Page 2 of 3). Statements and Options Allowed for Loading and Accessing VSAM Indexed Data Sets

File declaration <sup>1</sup>	Valid statements, with options you must include	Other options you can also include
SEQUENTIAL UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference);	EVENT(event-reference) and/or either KEY(expression) or KEYTO(reference)
	READ FILE(file-reference); <sup>2</sup>	EVENT(event-reference) and/or IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	EVENT(event reference)
	REWRITE FILE(file-reference) FROM(reference);	EVENT(event-reference) and/or KEY(expression)
	DELETE FILE(file-reference); <sup>5</sup>	KEY(expression) and/or EVENT(event-reference)
DIRECT <sup>4</sup> INPUT BUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	
	READ FILE(file-reference) SET(pointer-reference) KEY(expression);	
DIRECT <sup>4</sup> INPUT UNBUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	EVENT(event-reference)
DIRECT OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
DIRECT OUTPUT UNBUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	EVENT(event-reference)
DIRECT <sup>4</sup> UPDATE BUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	
	READ FILE(file-reference) SET(pointer-reference) KEY(expression);	
	REWRITE FILE(file-reference) FROM(reference) KEY(expression);	
	DELETE FILE(file-reference) KEY(expression); <sup>5</sup>	
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	

Table 37 (Page 3 of 3). Statements and Options Allowed for Loading and Accessing VSAM Indexed Data Sets

File declaration <sup>1</sup>	Valid statements, with options you must include	Other options you can also include
DIRECT <sup>4</sup> UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	EVENT(event-reference)
	REWRITE FILE(file-reference) FROM(reference) KEY(expression);	EVENT(event-reference)
	DELETE FILE(file-reference) KEY(expression); <sup>5</sup>	EVENT(event-reference)
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	EVENT(event-reference)

**Notes:**

1. The complete file declaration would include the attributes FILE and RECORD. If you use any of the options KEY, KEYFROM, or KEYTO, you must also include the attribute KEYED in the declaration. The EXCLUSIVE attribute for DIRECT INPUT or UPDATE files, the UNLOCK statement for DIRECT UPDATE files, or the NOLOCK option of the READ statement for DIRECT INPUT files are ignored if you use them for files associated with a VSAM KSDS.
2. The statement READ FILE(file-reference); is equivalent to the statement READ FILE(file-reference) IGNORE(1);
3. Do not associate a SEQUENTIAL OUTPUT file with a data set accessed via an alternate index.
4. Do not associate a DIRECT file with a data set accessed via a nonunique alternate index.
5. DELETE statements are not allowed for a file associated with an ESDS accessed via an alternate index.

## Loading a KSDS or Indexed ESDS

When a KSDS is being loaded, you must open the associated file for KEYED SEQUENTIAL OUTPUT. You must present the records in ascending key order, and you must use the KEYFROM option. Note that you must use the prime index for loading the data set; you cannot load a VSAM data set via an alternate index.

If a KSDS already contains some records, and you open the associated file with the SEQUENTIAL and OUTPUT attributes, you can only add records at the end of the data set. The rules given in the previous paragraph apply; in particular, the first record you present must have a key greater than the highest key present on the data set.

Figure 55 on page 235 shows the DEFINE command used to define a KSDS. The data set is given the name PLIVSAM.AJC2.BASE and defined as a KSDS because of the use of the INDEXED operand. The position of the keys within the record is defined in the KEYS operand.

Within the PL/I program, a KEYED SEQUENTIAL OUTPUT file is used with a WRITE...FROM...KEYFROM statement. The data is presented in ascending key order. A KSDS must be loaded in this manner.

The file is associated with the data set by a DD statement which uses the name given in the DEFINE command as the DSNNAME parameter.

```

//OPT9#12 JOB
// EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
  DEFINE CLUSTER -
    (NAME(PLIVSAM.AJC2.BASE) -
    VOLUMES(nnnnnn) -
    INDEXED -
    TRACKS(3 1) -
    KEYS(20 0) -
    RECORDSIZE(23 80))
/*
// EXEC IELICLG
//PLI.SYSIN DD *
  TELNOS: PROC OPTIONS(MAIN);

    DCL DIREC FILE RECORD SEQUENTIAL OUTPUT KEYED ENV(VSAM),
      CARD CHAR(80),
      NAME CHAR(20) DEF CARD POS(1),
      NUMBER CHAR(3) DEF CARD POS(21),
      OUTREC CHAR(23) DEF CARD POS(1),
      EOF BIT(1) INIT('0'B);

    ON ENDFILE(SYSIN) EOF='1'B;

    OPEN FILE(DIREC) OUTPUT;

    GET FILE(SYSIN) EDIT(CARD)(A(80));
    DO WHILE (-EOF);
    WRITE FILE(DIREC) FROM(OUTREC) KEYFROM(NAME);
    GET FILE(SYSIN) EDIT(CARD)(A(80));
    END;

    CLOSE FILE(DIREC);

    END TELNOS;
/*
//GO.DIREC DD DSNAME=PLIVSAM.AJC2.BASE,DISP=OLD
//GO.SYSIN DD *
ACTION,G.          162
BAKER,R.           152
BRAMLEY,O.H.      248
CHEESEMAN,D.      141
CORY,G.           336
ELLIOTT,D.        875
FIGGINS,S.        413
HARVEY,C.D.W.    205
HASTINGS,G.M.    391
KENDALL,J.G.     294
LANCASTER,W.R.   624
MILES,R.         233
NEWMAN,M.W.     450
PITT,W.H.        515
ROLF,D.E.        114
SHEERS,C.D.     241
SUTCLIFFE,M.    472
TAYLOR,G.C.     407
WILTON,L.W.     404
WINSTONE,E.M.   307
//

```

Figure 55. Defining and Loading a Key-Sequenced Data Set (KSDS)

## Using a SEQUENTIAL File to Access a KSDS or Indexed ESDS

You can open a SEQUENTIAL file that is used to access a KSDS with either the INPUT or the UPDATE attribute.

For READ statements without the KEY option, the records are recovered in ascending key order (or in descending key order if the BKWD option is used). You can obtain the key of a record recovered in this way by means of the KEYTO option.

If you use the KEY option, the record recovered by a READ statement is the one with the specified key. Such a READ statement positions the data set at the specified record; subsequent sequential reads will recover the following records in sequence.

WRITE statements with the KEYFROM option are allowed for KEYED SEQUENTIAL UPDATE files. You can make insertions anywhere in the data set, without respect to the position of any previous access. If you are accessing the data set via a unique index, the KEY condition is raised if an attempt is made to insert a record with the same key as a record that already exists on the data set. For a nonunique index, subsequent retrieval of records with the same key is in the order that they were added to the data set.

REWRITE statements with or without the KEY option are allowed for UPDATE files. If you use the KEY option, the record that is rewritten is the first record with the specified key; otherwise, it is the record that was accessed by the previous READ statement. When you rewrite a record using an alternate index, do not change the prime key of the record.

## Using a DIRECT File to Access a KSDS or Indexed ESDS

You can open a DIRECT file that is used to access an indexed VSAM data set with the INPUT, OUTPUT, or UPDATE attribute. Do not use a DIRECT file to access the data set via a nonunique index.

If you use a DIRECT OUTPUT file to add records to the data set, and if an attempt is made to insert a record with the same key as a record that already exists, the KEY condition is raised.

If you use a DIRECT INPUT or DIRECT UPDATE file, you can read, write, rewrite, or delete records in the same way as for a KEYED SEQUENTIAL file.

Figure 56 on page 237 shows one method by which a KSDS can be updated using the prime index.

---

```

//OPT9#13 JOB
//STEP1 EXEC IEL1CLG
//PLI.SYSIN DD *
DIRUPDT: PROC OPTIONS(MAIN);

    DCL DIREC FILE RECORD KEYED ENV(VSAM),
        ONCODE BUILTIN,
        OUTREC CHAR(23),
        NUMBER CHAR(3) DEF OUTREC POS(21),
        NAME CHAR(20) DEF OUTREC,
        CODE CHAR(1),
        EOF BIT(1) INIT('0'B);

ON ENDFILE(SYSIN) EOF='1'B;

ON KEY(DIREC) BEGIN;
    IF ONCODE=51 THEN PUT FILE(SYSPRINT) SKIP EDIT
        ('NOT FOUND: ',NAME)(A(15),A);
    IF ONCODE=52 THEN PUT FILE(SYSPRINT) SKIP EDIT
        ('DUPLICATE: ',NAME)(A(15),A);
    END;

OPEN FILE(DIREC) DIRECT UPDATE;

GET FILE(SYSIN) EDIT (NAME,NUMBER,CODE)
    (COLUMN(1),A(20),A(3),A(1));
DO WHILE (-EOF);
    PUT FILE(SYSPRINT) SKIP EDIT (' ',NAME,'#',NUMBER,' ',CODE)
    (A(1),A(20),A(1),A(3),A(1),A(1));
    SELECT (CODE);
        WHEN('A') WRITE FILE(DIREC) FROM(OUTREC) KEYFROM(NAME);
        WHEN('C') REWRITE FILE(DIREC) FROM(OUTREC) KEY(NAME);
        WHEN('D') DELETE FILE(DIREC) KEY(NAME);
        OTHERWISE PUT FILE(SYSPRINT) SKIP EDIT
            ('INVALID CODE: ',NAME) (A(15),A);
    END;
GET FILE(SYSIN) EDIT (NAME,NUMBER,CODE)
    (COLUMN(1),A(20),A(3),A(1));
END;

```

---

Figure 56 (Part 1 of 2). Updating a KSDS

---

```

CLOSE FILE(DIREC);
PUT FILE(SYSPRINT) PAGE;
OPEN FILE(DIREC) SEQUENTIAL INPUT;

EOF='0'B;
ON ENDFILE(DIREC) EOF='1'B;

READ FILE(DIREC) INTO(OUTREC);
DO WHILE(¬EOF);
PUT FILE(SYSPRINT) SKIP EDIT(OUTREC) (A);
READ FILE(DIREC) INTO(OUTREC);
END;
CLOSE FILE(DIREC);
END DIRUPDT;
/*
//GO.DIREC DD DSNAME=PLIVSAM.AJC2.BASE,DISP=OLD
//GO.SYSIN DD *
NEWMAN,M.W.      516C
GOODFELLOW,D.T. 889A
MILES,R.         D
HARVEY,C.D.W.   209A
BARTLETT,S.G.   183A
CORY,G.         D
READ,K.M.       001A
PITT,W.H.
ROLF,D.F.       D
ELLIOTT,D.     291C
HASTINGS,G.M.  D
BRAMLEY,O.H.   439C
/*

```

---

Figure 56 (Part 2 of 2). Updating a KSDS

A DIRECT update file is used and the data is altered according to a code that is passed in the records in the file SYSIN:

- A Add a new record
- C Change the number of an existing name
- D Delete a record

At the label NEXT, the name, number, and code are read in and action taken according to the value of the code. A KEY ON-unit is used to handle any incorrect keys. When the updating is finished (at the label PRINT), the file DIREC is closed and reopened with the attributes SEQUENTIAL INPUT. The file is then read sequentially and printed.

The file is associated with the data set by a DD statement that uses the DSNAME PLIVSAM.AJC2.BASE defined in the Access Method Services DEFINE CLUSTER command in Figure 55 on page 235.

**Methods of Updating a KSDS:** There are a number of methods of updating a KSDS. The method shown using a DIRECT file is suitable for the data as it is shown in the example. If the data had been presented in ascending key order (or even something approaching it), performance might have been improved by use of the SKIP ENVIRONMENT option. For mass sequential insertion, use a KEYED SEQUENTIAL UPDATE file. This gives faster performance because the data is written onto the data set only when strictly necessary and not after every write statement, and because the balance of free space within the data set is retained.

Statements to achieve effective mass sequential insertion are:

```
DCL DIREC KEYED SEQUENTIAL UPDATE
  ENV(VSAM);
WRITE FILE(DIREC) FROM(OUTREC)
  KEYFROM(NAME);
```

The PL/I input/output routines detect that the keys are in sequence and make the correct requests to VSAM. If the keys are not in sequence, this too is detected and no error occurs, although the performance advantage is lost. VSAM provides three methods of insertion as shown in Table 38.

Table 38. VSAM Methods of Insertion into a KSDS

Method	Requirements	Freespace	When written onto data set	PL/I attributes required
SEQ	Keys in sequence	Kept	Only when necessary	KEYED SEQUENTIAL UPDATE
SKP	Keys in sequence	Used	Only when necessary	KEYED SEQUENTIAL UPDATE ENV(VSAM SKIP)
DIR	Keys in any order	Used	After every statement	DIRECT
DIR(MACRF=SIS)	Keys in any order	Kept	After every statement	DIRECT ENV(VSAM SIS)

SKIP means that you must follow the sequence but that you can omit records. You do not need to maintain absolute sequence or order if SEQ or SKIP is used. The PL/I routines determine which type of request to make to VSAM for each statement, first checking the keys to determine which would be appropriate. The retention of free space ensures that the structure of the data set at the point of mass sequential insertion is not destroyed, enabling you to make further normal alterations in that area without loss of performance. To preserve free space balance when you require immediate writing of the data set during mass sequential insertion, as it can be on interactive systems, use the SIS ENVIRONMENT option with DIRECT files.

## Alternate Indexes for KSDSs or Indexed ESDSs

Alternate indexes allow you to access KSDSs or indexed ESDSs in various ways, using either unique or nonunique keys.

### Unique Key Alternate Index Path

Figure 57 on page 240 shows the creation of a unique key alternative index path for the ESDS defined and loaded in Figure 53 on page 231. Using this path, the data set is indexed by the name of the child in the first 15 bytes of the record.

Three Access Method Services commands are used. These are:

#### DEFINE ALTERNATEINDEX

defines the alternate index as a data set to VSAM.

#### BLDINDEX

places the pointers to the relevant records in the alternate index.

#### DEFINE PATH

defines an entity that can be associated with a PL/I file in a DD statement.

DD statements are required for the INFILE and OUTFILE operands of BLDINDEX and for the sort files. Make sure that you specify the correct names at the various points.

---

```
//OPT9#9 JOB
//STEP1 EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
        DEFINE ALTERNATEINDEX -
            (NAME(PLIVSAM.AJC1.ALPHIND) -
             VOLUMES(nnnnnn) -
             TRACKS(4 1) -
             KEYS(15 0) -
             RECORDSIZE(20 40) -
             UNIQUEKEY -
             RELATE(PLIVSAM.AJC1.BASE))
/*
//STEP2 EXEC PGM=IDCAMS,REGION=512K
//DD1 DD DSN=PLIVSAM.AJC1.BASE,DISP=SHR
//DD2 DD DSN=PLIVSAM.AJC1.ALPHIND,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSIN DD *
        BLDINDEX INFILE(DD1) OUTFILE(DD2)
        DEFINE PATH -
            (NAME(PLIVSAM.AJC1.ALPHPATH) -
             PATHENTRY(PLIVSAM.AJC1.ALPHIND))
//
```

---

Figure 57. Creating a Unique Key Alternate Index Path for an ESDS

### Nonunique Key Alternate Index Path

Figure 58 on page 241 shows the creation of a nonunique key alternate index path for an ESDS. The alternate index enables the data to be selected by the sex of the children. This enables the girls or the boys to be accessed separately and every member of each group to be accessed by use of the key.

The three Access Method Services commands and the DD statement are as described in “Unique Key Alternate Index Path” on page 239. The fact that the index has nonunique keys is specified by the use of the NONUNIQUEKEY operand. When creating an index with nonunique keys, be careful to ensure that the RECORDSIZE you specify is large enough. In a nonunique alternate index, each alternate index record contains pointers to all the records that have the associated alternate index key. The pointer takes the form of an RBA for an ESDS and the prime key for a KSDS. When a large number of records might have the same key, a large record is required.

---

```

//OPT9#10 JOB
//STEP1 EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
/*care must be taken with record size */
DEFINE ALTERNATEINDEX -
  (NAME(PLIVSAM.AJC1.SEXIND) -
  VOLUMES(nnnnnn) -
  TRACKS(4 1) -
  KEYS(1 37) -
  NONUNIQUEKEY -
  RELATE(PLIVSAM.AJC1.BASE) -
  RECORDSIZE(20 400))
/*
//STEP2 EXEC PGM=IDCAMS,REGION=512K
//DD1 DD DSNAME=PLIVSAM.AJC1.BASE,DISP=OLD
//DD2 DD DSNAME=PLIVSAM.AJC1.SEXIND,DISP=OLD
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
BLDINDEX INFILE(DD1) OUTFILE(DD2)
DEFINE PATH -
  (NAME(PLIVSAM.AJC1.SEXPATH) -
  PATHENTRY(PLIVSAM.AJC1.SEXIND))
//

```

---

Figure 58. Creating a Nonunique Key Alternate Index Path for an ESDS

Figure 59 on page 242 shows the creation of a unique key alternate index path for a KSDS. The data set is indexed by the telephone number, enabling the number to be used as a key to discover the name of person on that extension. The fact that keys are to be unique is specified by `UNIQUEKEY`. Also, the data set will be able to be listed in numerical order to show which numbers are not used. Three Access Method Services commands are used:

#### **DEFINE ALTERNATEINDEX**

defines the data set that will hold the alternate index data.

#### **BLDINDEX**

places the pointers to the relevant records in the alternate index.

#### **DEFINE PATH**

defines the entity that can be associated with a PL/I file in a DD statement.

DD statements are required for the `INFILE` and `OUTFILE` of `BLDINDEX` and for the sort files. Be careful not to confuse the names involved.

---

```

//OPT9#14 JOB
//STEP1 EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
        DEFINE ALTERNATEINDEX -
            (NAME(PLIVSAM.AJC2.NUMIND) -
             VOLUMES(nnnnnn) -
             TRACKS(4 4) -
             KEYS(3 20) -
             RELATE(PLIVSAM.AJC2.BASE) -
             UNIQUEKEY -
             RECORDSIZE(24 48))

/*
//STEP2 EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT DD SYSOUT=A
//DD1 DD DSN=PLIVSAM.AJC2.BASE,DISP=OLD
//DD2 DD DSN=PLIVSAM.AJC2.NUMIND,DISP=OLD
//SYSIN DD *
        BLDINDEX INFILE(DD1) OUTFILE(DD2)

        DEFINE PATH -
            (NAME(PLIVSAM.AJC2.NUMPATH) -
             PATHENTRY(PLIVSAM.AJC2.NUMIND))

//

```

---

Figure 59. Creating an Alternate Index Path for a KSDS

When creating an alternate index with a unique key, you should ensure that no further records could be included with the same alternative key. In practice, a unique key alternate index would not be entirely satisfactory for a telephone directory as it would not allow two people to have the same number. Similarly, the prime key would prevent one person having two numbers. A solution would be to have an ESDS with two nonunique key alternate indexes, or to restructure the data format to allow more than one number per person and to have a nonunique key alternate index for the numbers. See Figure 57 on page 240 for an example of creation of an alternate index with nonunique keys.

### Detecting Nonunique Alternate Index Keys

If you are accessing a VSAM data set by means of an alternate index path, the presence of nonunique keys can be detected by means of the SAMEKEY built-in function. After each retrieval, SAMEKEY indicates whether any further records exist with the same alternate index key as the record just retrieved. Hence, it is possible to stop at the last of a series of records with nonunique keys without having to read beyond the last record. SAMEKEY (file-reference) returns '1'B if the input/output statement has completed successfully and the accessed record is followed by another with the same key; otherwise, it returns '0'B.

### Using Alternate Indexes with ESDSs

Figure 60 on page 244 shows the use of alternate indexes and backward reading on an ESDS. The program has four files:

BASEFLE reads the base data set forward.

BACKFLE reads the base data set backward.

ALPHFLE is the alphabetic alternate index path indexing the children by name.

SEXFILE is the alternate index path that corresponds to the sex of the children.

There are DD statements for all the files. They connect BASEFLE and BACKFLE to the base data set by specifying the name of the base data set in the DSNNAME parameter, and connect ALPHFLE and SEXFLE by specifying the names of the paths given in Figure 57 on page 240 and Figure 58 on page 241.

The program uses SEQUENTIAL files to access the data and print it first in the normal order, then in the reverse order. At the label AGEQUERY, a DIRECT file is used to read the data associated with an alternate index key in the unique alternate index.

Finally, at the label SPRINT, a KEYED SEQUENTIAL file is used to print a list of the females in the family, using the nonunique key alternate index path. The SAMEKEY built-in function is used to read all the records with the same key. The names of the females will be accessed in the order in which they were originally entered. This will happen whether the file is read forward or backward. For a nonunique key path, the BKWD option only affects the order in which the keys are read; the order of items with the same key remains the same as it is when the file is read forward.

**Deletion:** At the end of the example, the Access Method Services DELETE command is used to delete the base data set. When this is done, the associated alternate indexes and paths will also be deleted.

### Using Alternate Indexes with KSDSs

Figure 61 on page 246 shows the use of a path with a unique alternate index key to update a KSDS and then to access and print it in the order of the alternate index.

The alternate index path is associated with the PL/I file by a DD statement that specifies the name of the path (PLIVSAM.AJC2.NUMPATH, given in the DEFINE PATH command in Figure 59 on page 242) as the DSNAME.

In the first section of the program, a DIRECT OUTPUT file is used to insert a new record using the alternate index key. Note that any alteration made with an alternate index must not alter the prime key or the alternate index key of access of an existing record. Also, the alteration must not add a duplicate key in the prime index or any unique key alternate index.

In the second section of the program (at the label PRINTIT), the data set is read in the order of the alternate index keys using a SEQUENTIAL INPUT file. It is then printed onto SYSPRINT.

---

```

//OPT9#15 JOB
//STEP1 EXEC IEL1CLG
//PLI.SYSIN DD *
READIT: PROC OPTIONS(MAIN);
  DCL BASEFLE FILE SEQUENTIAL INPUT ENV(VSAM),
        /*File to read base data set forward */
  BACKFLE FILE SEQUENTIAL INPUT ENV(VSAM BKWD),
        /*File to read base data set backward */
  ALPHFLE FILE DIRECT INPUT ENV(VSAM),
        /*File to access via unique alternate index path */
  SEXFILE FILE KEYED SEQUENTIAL INPUT ENV(VSAM),
        /*File to access via nonunique alternate index path */
  STRING CHAR(80), /*String to be read into */
  1 STRUC DEF (STRING),
    2 NAME CHAR(25),
    2 DATE_OF_BIRTH CHAR(2),
    2 FILL CHAR(10),
    2 SEX CHAR(1);
  DCL NAMEHOLD CHAR(25),SAMEKEY BUILTIN;
  DCL EOF BIT(1) INIT('0'B);

  /*Print out the family eldest first*/

  ON ENDFILE(BASEFLE) EOF='1'B;
  PUT EDIT('FAMILY ELDEST FIRST')(A);
  READ FILE(BASEFLE) INTO (STRING);
  DO WHILE(-EOF);
    PUT SKIP EDIT(STRING)(A);
    READ FILE(BASEFLE) INTO (STRING);
  END;
  CLOSE FILE(BASEFLE);
  PUT SKIP(2);
  /*Close before using data set from other file not
  necessary but good practice to prevent potential
  problems*/

  EOF='0'B;
  ON ENDFILE(BACKFLE) EOF='1'B;
  PUT SKIP(3) EDIT('FAMILY YOUNGEST FIRST')(A);
  READ FILE(BACKFLE) INTO(STRING);
  DO WHILE(-EOF);
    PUT SKIP EDIT(STRING)(A);
    READ FILE(BACKFLE) INTO (STRING);
  END;
  CLOSE FILE(BACKFLE);
  PUT SKIP(2);

  /*Print date of birth of child specified in the file
  SYSIN*/
  ON KEY(ALPHFLE) BEGIN;
    PUT SKIP EDIT
      (NAMEHOLD,' NOT A MEMBER OF THE SMITH FAMILY')(A);
    GO TO SPRINT;
  END;

```

---

Figure 60 (Part 1 of 2). Alternate Index Paths and Backward Reading with an ESDS

---

```

AGEQUERY:
  EOF='0'B;
  ON ENDFILE(SYSIN) EOF='1'B;
  GET SKIP EDIT(NAMEHOLD)(A(25));
  DO WHILE(-EOF);
    READ FILE(ALPHFLE) INTO (STRING) KEY(NAMEHOLD);
    PUT SKIP (2) EDIT(NAMEHOLD,' WAS BORN IN ',
      DATE_OF_BIRTH)(A,X(1),A,X(1),A);
    GET SKIP EDIT(NAMEHOLD)(A(25));
  END;
SPRINT:
  CLOSE FILE(ALPHFLE);
  PUT SKIP(1);

/*Use the alternate index to print out all the females in the
family*/
  ON ENDFILE(SEXFILE) GOTO FINITO;
  PUT SKIP(2) EDIT('ALL THE FEMALES')(A);
  READ FILE(SEXFILE) INTO (STRING) KEY('F');
  PUT SKIP EDIT(STRING)(A);
  DO WHILE(SAMEKEY(SEXFILE));
    READ FILE(SEXFILE) INTO (STRING);
    PUT SKIP EDIT(STRING)(A);
  END;

FINITO:
  END;

/*
//GO.BASEFLE DD DSN=PLIVSAM.AJC1.BASE,DISP=SHR
//GO.BACKFLE DD DSN=PLIVSAM.AJC1.BASE,DISP=SHR
//GO.ALPHFLE DD DSN=PLIVSAM.AJC1.ALPHPATH,DISP=SHR
//GO.SEXFILE DD DSN=PLIVSAM.AJC1.SEXPATH,DISP=SHR
//GO.SYSIN DD *
ANDY
/*
//STEP2 EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
DELETE -
        PLIVSAM.AJC1.BASE
//

```

---

Figure 60 (Part 2 of 2). Alternate Index Paths and Backward Reading with an ESDS

---

```

//OPT9#16 JOB
//STEP1 EXEC IEL1CLG,REGION.GO=256K
//PLI.SYSIN DD *
ALTER: PROC OPTIONS(MAIN);
  DCL NUMFLE1 FILE RECORD DIRECT OUTPUT ENV(VSAM),
  NUMFLE2 FILE RECORD SEQUENTIAL INPUT ENV(VSAM),
  IN FILE RECORD,
  STRING CHAR(80),
  NAME CHAR(20) DEF STRING,
  NUMBER CHAR(3) DEF STRING POS(21),
  DATA CHAR(23) DEF STRING,
  EOF BIT(1) INIT('0'B);

ON KEY (NUMFLE1) BEGIN;
  PUT SKIP EDIT('DUPLICATE NUMBER')(A);
END;

ON ENDFILE(IN) EOF='1'B;

READ FILE(IN) INTO (STRING);
DO WHILE(-EOF);
  PUT FILE(SYSPRINT) SKIP EDIT (STRING) (A);
  WRITE FILE(NUMFLE1) FROM (STRING) KEYFROM(NUMBER);
  READ FILE(IN) INTO (STRING);
END;

CLOSE FILE(NUMFLE1);

EOF='0'B;
ON ENDFILE(NUMFLE2) EOF='1'B;

READ FILE(NUMFLE2) INTO (STRING);
DO WHILE(-EOF);
  PUT SKIP EDIT(DATA)(A);
  READ FILE(NUMFLE2) INTO (STRING);
END;

PUT SKIP(3) EDIT('****SO ENDS THE PHONE DIRECTORY****')(A);
END;
/*
//GO.IN DD *
RIERA L 123
/*
//NUMFLE1 DD DSN=PLIVSAM.AJC2.NUMPATH,DISP=OLD
//NUMFLE2 DD DSN=PLIVSAM.AJC2.NUMPATH,DISP=OLD
//STEP2 EXEC PGM=IDCAMS,COND=EVEN
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
DELETE -
PLIVSAM.AJC2.BASE
//

```

---

Figure 61. Using a Unique Alternate Index Path to Access a KSDS

## Relative-Record Data Sets

The statements and options allowed for VSAM relative-record data sets (RRDS) are shown in Table 39.

Table 39 (Page 1 of 3). Statements and Options Allowed for Loading and Accessing VSAM Relative-Record Data Sets

File declaration <sup>1</sup>	Valid statements, with options you must include	Other options you can also include
SEQUENTIAL OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference);	KEYFROM(expression) or KEYTO(reference)
	LOCATE based-variable FILE(file-reference);	SET(pointer-reference)
SEQUENTIAL OUTPUT UNBUFFERED	WRITE FILE(file-reference) FROM(reference);	EVENT(event-reference) and/or either KEYFROM(expression) or KEYTO(reference)
SEQUENTIAL INPUT BUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference); <sup>2</sup>	IGNORE(expression)
SEQUENTIAL INPUT UNBUFFERED	READ FILE(file-reference) INTO(reference);	EVENT(event-reference) and/or either KEY(expression) or KEYTO(reference)
	READ FILE(file-reference); <sup>2</sup>	EVENT(event-reference) and/or IGNORE(expression)
SEQUENTIAL UPDATE BUFFERED	READ FILE(file-reference) INTO(reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference) SET(pointer-reference);	KEY(expression) or KEYTO(reference)
	READ FILE(file-reference); <sup>2</sup>	IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference);	KEYFROM(expression) or KEYTO(reference)
	REWRITE FILE(file-reference);	FROM(reference) and/or KEY(expression)
	DELETE FILE(file-reference);	KEY(expression)

Table 39 (Page 2 of 3). Statements and Options Allowed for Loading and Accessing VSAM Relative-Record Data Sets

<b>File declaration<sup>1</sup></b>	<b>Valid statements, with options you must include</b>	<b>Other options you can also include</b>
SEQUENTIAL UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference);	EVENT(event-reference) and/or either KEY(expression) or KEYTO(reference)
	READ FILE(file-expression); <sup>2</sup>	EVENT(event-reference) and/or IGNORE(expression)
	WRITE FILE(file-reference) FROM(reference);	EVENT(event-reference) and/or either KEYFROM(expression) or KEYTO(reference)
	REWRITE FILE(file-reference) FROM(reference);	EVENT(event-reference) and/or KEY(expression)
	DELETE FILE(file-reference);	EVENT(event-reference) and/or KEY(expression)
DIRECT OUTPUT BUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	
DIRECT OUTPUT UNBUFFERED	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	EVENT(event-reference)
DIRECT INPUT BUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	
	READ FILE(file-reference) SET(pointer-reference) KEY(expression);	
DIRECT INPUT UNBUFFERED	READ FILE(file-reference) KEY(expression);	EVENT(event-reference)
DIRECT UPDATE BUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	
	READ FILE(file-reference) SET(pointer-reference) KEY(expression);	
	REWRITE FILE(file-reference) FROM(reference) KEY(expression);	
	DELETE FILE(file-reference) KEY(expression);	
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	

Table 39 (Page 3 of 3). Statements and Options Allowed for Loading and Accessing VSAM Relative-Record Data Sets

File declaration <sup>1</sup>	Valid statements, with options you must include	Other options you can also include
DIRECT UPDATE UNBUFFERED	READ FILE(file-reference) INTO(reference) KEY(expression);	EVENT(event-reference)
	REWRITE FILE(file-reference) FROM(reference) KEY(expression);	EVENT(event-reference)
	DELETE FILE(file-reference) KEY(expression);	EVENT(event-reference)
	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);	EVENT(event-reference)

**Notes:**

1. The complete file declaration would include the attributes FILE and RECORD. If you use any of the options KEY, KEYFROM, or KEYTO, your declaration must also include the attribute KEYED.  
The EXCLUSIVE attribute for DIRECT INPUT or UPDATE files, the UNLOCK statement for DIRECT UPDATE files, or the NOLOCK option of the READ statement for DIRECT INPUT files are ignored if you use them for files associated with a VSAM RRDS.
2. The statement READ FILE(file-reference); is equivalent to the statement READ FILE(file-reference) IGNORE(1);

## Loading an RRDS

When an RRDS is being loaded, you must open the associated file for OUTPUT. Use either a DIRECT or a SEQUENTIAL file.

For a DIRECT OUTPUT file, each record is placed in the position specified by the relative record number (or key) in the KEYFROM option of the WRITE statement (see “Keys for VSAM Data Sets” on page 215).

For a SEQUENTIAL OUTPUT file, use WRITE statements with or without the KEYFROM option. If you specify the KEYFROM option, the record is placed in the specified slot; if you omit it, the record is placed in the slot following the current position. There is no requirement for the records to be presented in ascending relative record number order. If you omit the KEYFROM option, you can obtain the relative record number of the written record by means of the KEYTO option.

If you want to load an RRDS sequentially, without use of the KEYFROM or KEYTO options, your file is not required to have the KEYED attribute.

It is an error to attempt to load a record into a position that already contains a record: if you use the KEYFROM option, the KEY condition is raised; if you omit it, the ERROR condition is raised.

In Figure 62 on page 250, the data set is defined with a DEFINE CLUSTER command and given the name PLIVSAM.AJC3.BASE. The fact that it is an RRDS is determined by the NUMBERED keyword. In the PL/I program, it is loaded with a DIRECT OUTPUT file and a WRITE...FROM...KEYFROM statement is used.

If the data had been in order and the keys in sequence, it would have been possible to use a SEQUENTIAL file and write into the data set from the start. The records would then have been placed in the next available slot and given the appropriate number. The number of the key for each record could have been returned using the KEYTO option.

The PL/I file is associated with the data set by the DD statement, which uses as the DSNAME the name given in the DEFINE CLUSTER command.

---

```
//OPT9#17 JOB
//STEP1 EXEC PGM=IDCAMS,REGION=512K
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
    DEFINE CLUSTER -
        (NAME(PLIVSAM.AJC3.BASE) -
        VOLUMES(nnnnnn) -
        NUMBERED -
        TRACKS(2 2) -
        RECORDSIZE(20 20))
/*
//STEP2 EXEC IELICLG
//PLI.SYSIN DD *
CRR1: PROC OPTIONS(MAIN);
    DCL NOS FILE RECORD OUTPUT DIRECT KEYED ENV(VSAM),
        CARD CHAR(80),
        NAME CHAR(20) DEF CARD,
        NUMBER CHAR(2) DEF CARD POS(21),
        IOFIELD CHAR(20),
        EOF BIT(1) INIT('0'B);
    ON ENDFILE (SYSIN) EOF='1'B;
    OPEN FILE(NOS);
    GET FILE(SYSIN) EDIT(CARD)(A(80));
    DO WHILE (~EOF);
        PUT FILE(SYSPRINT) SKIP EDIT (CARD) (A);
        IOFIELD=NAME;
        WRITE FILE(NOS) FROM(IOFIELD) KEYFROM(NUMBER);
        GET FILE(SYSIN) EDIT(CARD)(A(80));
    END;
    CLOSE FILE(NOS);
END CRR1;
```

---

Figure 62 (Part 1 of 2). Defining and Loading a Relative-Record Data Set (RRDS)

---

```

/*
//GO.NOS DD DSN=PLIVSAM.AJC3.BASE,DISP=OLD
//GO.SYSIN DD *
ACTION,G.          12
BAKER,R.           13
BRAMLEY,O.H.      28
CHEESNAME,L.      11
CORY,G.           36
ELLIOTT,D.        85
FIGGINS.E.S.      43
HARVEY,C.D.W.    25
HASTINGS,G.M.     31
KENDALL,J.G.      24
LANCASTER,W.R.   64
MILES,R.          23
NEWMAN,M.W.      40
PITT,W.H.         55
ROLF,D.E.         14
SHEERS,C.D.       21
SURCLIFFE,M.     42
TAYLOR,G.C.       47
WILTON,L.W.       44
WINSTONE,E.M.    37
//

```

---

Figure 62 (Part 2 of 2). Defining and Loading a Relative-Record Data Set (RRDS)

## Using a SEQUENTIAL File to Access an RRDS

You can open a SEQUENTIAL file that is used to access an RRDS with either the INPUT or the UPDATE attribute. If you use any of the options KEY, KEYTO, or KEYFROM, your file must also have the KEYED attribute.

For READ statements without the KEY option, the records are recovered in ascending relative record number order. Any empty slots in the data set are skipped.

If you use the KEY option, the record recovered by a READ statement is the one with the relative record number you specify. Such a READ statement positions the data set at the specified record; subsequent sequential reads will recover the following records in sequence.

WRITE statements with or without the KEYFROM option are allowed for KEYED SEQUENTIAL UPDATE files. You can make insertions anywhere in the data set, regardless of the position of any previous access. For WRITE with the KEYFROM option, the KEY condition is raised if an attempt is made to insert a record with the same relative record number as a record that already exists on the data set. If you omit the KEYFROM option, an attempt is made to write the record in the next slot, relative to the current position. The ERROR condition is raised if this slot is not empty.

You can use the KEYTO option to recover the key of a record that is added by means of a WRITE statement without the KEYFROM option.

REWRITE statements, with or without the KEY option, are allowed for UPDATE files. If you use the KEY option, the record that is rewritten is the record with the relative record number you specify; otherwise, it is the record that was accessed by the previous READ statement.

DELETE statements, with or without the KEY option, can be used to delete records from the dataset.

## Using a DIRECT File to Access an RRDS

A DIRECT file used to access an RRDS can have the OUTPUT, INPUT, or UPDATE attribute. You can read, write, rewrite, or delete records exactly as though a KEYED SEQUENTIAL file were used.

Figure 63 on page 253 shows an RRDS being updated. A DIRECT UPDATE file is used and new records are written by key. There is no need to check for the records being empty, because the empty records are not available under VSAM.

In the second half of the program, starting at the label PRINT, the updated file is printed out. Again there is no need to check for the empty records as there is in REGIONAL(1).

The PL/I file is associated with the data sets by a DD statement that specifies the DSNAME PLIVSAM.AJC3.BASE, the name given in the DEFINE CLUSTER command in Figure 63 on page 253.

At the end of the example, the DELETE command is used to delete the data set.

---

```

/** NOTE: WITH A WRITE STATEMENT AFTER THE DELETE FILE STATEMENT,
/**      A "DUPLICATE" MESSAGE IS EXPECTED FOR CODE 'C' ITEMS
/**      WHOSE NEWNO CORRESPONDS TO AN EXISTING NUMBER IN THE LIST,
/**      FOR EXAMPLE, ELLIOT.
/**      WITH A REWRITE STATEMENT AFTER THE DELETE FILE STATEMENT,
/**      A "NOT FOUND" MESSAGE IS EXPECTED FOR CODE 'C' ITEMS
/**      WHOSE NEWNO DOES NOT CORRESPOND TO AN EXISTING NUMBER IN
/**      THE LIST, FOR EXAMPLE, NEWMAN AND BRAMLEY.
//OPT9#18 JOB
//STEP1 EXEC IEL1CLG
//PLI.SYSIN DD *
ACR1: PROC OPTIONS(MAIN);
      DCL NOS FILE RECORD KEYED ENV(VSAM),NAME CHAR(20),
      (NEWNO,OLDNO) CHAR(2),CODE CHAR(1),IOFIELD CHAR(20),
      BYTE CHAR(1) DEF IOFIELD, EOF BIT(1) INIT('0'B),
      ONCODE BUILTIN;
ON ENDFILE(SYSIN) EOF='1'B;
OPEN FILE(NOS) DIRECT UPDATE;
ON KEY(NOS) BEGIN;
  IF ONCODE=51 THEN PUT FILE(SYSPRINT) SKIP EDIT
    ('NOT FOUND:',NAME)(A(15),A);
  IF ONCODE=52 THEN PUT FILE(SYSPRINT) SKIP EDIT
    ('DUPLICATE:',NAME)(A(15),A);
END;
GET FILE(SYSIN) EDIT(NAME,NEWNO,OLDNO,CODE)
  (COLUMN(1),A(20),A(2),A(2),A(1));
DO WHILE (-EOF);
PUT FILE(SYSPRINT) SKIP EDIT (' ',NAME,'#',NEWNO,OLDNO,' ',CODE)
  (A(1),A(20),A(1),2(A(2)),X(5),2(A(1)));
SELECT(CODE);
  WHEN('A') WRITE FILE(NOS) KEYFROM(NEWNO) FROM(NAME);
  WHEN('C') DO;
    DELETE FILE(NOS) KEY(OLDNO);
    WRITE FILE(NOS) KEYFROM(NEWNO) FROM(NAME);
  END;
  WHEN('D') DELETE FILE(NOS) KEY(OLDNO);
  OTHERWISE PUT FILE(SYSPRINT) SKIP EDIT
    ('INVALID CODE: ',NAME)(A(15),A);
END;

```

---

Figure 63 (Part 1 of 2). Updating an RRDS

---

```

GET FILE(SYSIN) EDIT(NAME,NEWNO,OLDNO,CODE)
(COLUMN(1),A(20),A(2),A(2),A(1));
END;
CLOSE FILE(NOS);
PRINT:
PUT FILE(SYSPRINT) PAGE;
OPEN FILE(NOS) SEQUENTIAL INPUT;
EOF='0'B;
ON ENDFILE(NOS) EOF='1'B;
READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
DO WHILE (~EOF);
PUT FILE(SYSPRINT) SKIP EDIT(NEWNO,IOFIELD)(A(5),A);
READ FILE(NOS) INTO(IOFIELD) KEYTO(NEWNO);
END;
CLOSE FILE(NOS);
END ACR1;
/*
//GO.NOS DD DSN=PLIVSAM.AJC3.BASE,DISP=OLD
//GO.SYSIN DD *
NEWMAN,M.W. 5640C
GOODFELLOW,D.T. 89 A
MILES,R. 23D
HARVEY,C.D.W. 29 A
BARTLETT,S.G. 13 A
CORY,G. 36D
READ,K.M. 01 A
PITT,W.H. 55
ROLF,D.F. 14D
ELLIOTT,D. 4285C
HASTINGS,G.M. 31D
BRAMLEY,O.H. 4928C
//STEP3 EXEC PGM=IDCAMS,REGION=512K,COND=EVEN
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
DELETE -
PLIVSAM.AJC3.BASE
//

```

---

Figure 63 (Part 2 of 2). Updating an RRDS

---

## Chapter 12. Defining and Using Teleprocessing Data Sets

Teleprocessing in PL/I is supported by record-oriented data transmission using the Telecommunications Access Method (TCAM) and PL/I files declared with the TRANSIENT attribute. A teleprocessing data set is a queue of messages originating from or destined for remote terminals (or application programs). A PL/I TRANSIENT file allows a PL/I program to access a teleprocessing data set as an INPUT file for retrieving messages or as an OUTPUT file for writing messages.

In a teleprocessing system using TCAM, the user must write a message control program (MCP) and can write one or more message processing programs (TCAM MPPs). The MCP is part of TCAM and must be written in assembler language using macros supplied by TCAM. The TCAM MPPs are application programs and can be written in PL/I.

This section briefly covers the message control program (MCP) and the message processing program (TCAM MPP). It also covers teleprocessing organization, ENVIRONMENT options for teleprocessing, and condition handling for teleprocessing.

A TCAM overview is given in *OS/VS TCAM Concepts and Applications*. If you want more detailed information about TCAM programming facilities, see the *ACF TCAM Application Programmer's Guide*.

TCAM is not available under VM using PL/I.

### Message Control Program (MCP)

A TCAM message control program (MCP) controls the routing of messages originating from and destined for the remote terminals and message processing programs in your TCAM installation. Each origin or destination associated with a message is identified by a name known in the MCP, and carried within the message. The MCP routes messages to and from message processing programs and terminals by means of in-storage queues. The queues can also be on disk storage when the in-storage queue is full. This support is provided by TCAM. TCAM queues can also be simulated by sequential data sets on direct-access devices; however, the data sets cannot be accessed by your PL/I program, since PL/I supports only the use of queues.

A message can be transmitted in one of several formats, only two of which are supported by PL/I. You specify the message format in the MCP, and also in your PL/I program by means of the ENVIRONMENT attribute, described later in this section.

**Note for System Programmers:** Of the several message formats allowed by a TCAM MCP, PL/I supports those represented by:

- DCBOPTCD=WUC,DCBRECFM=V for PL/I ENVIRONMENT option TP(M)
- DCBOPTCD=WC,DCBRECFM=V for PL/I ENVIRONMENT option TP(R).

## TCAM Message Processing Program (TCAM MPP)

A message processing program (TCAM MPP) is an application program that retrieves messages from TCAM queues and/or writes messages to TCAM queues. A TCAM MPP allows you to provide data to a problem program from a terminal and to receive output from the program with a minimum of delay. You can write TCAM MPPs in PL/I; they can perform other data processing functions in addition to teleprocessing.

A TCAM MPP for reading or writing TCAM queues is not mandatory for teleprocessing installations. If the messages you transmit are not processed, because they are simply switched between terminals, then a TCAM MPP is not required.

The following sections describe PL/I teleprocessing data sets and the PL/I language features that you use to write TCAM MPPs.

## Teleprocessing Organization

A teleprocessing data set is a queue of messages that constitutes the input to a PL/I message processing program. You write and retrieve the messages sequentially. You use keys to identify the terminal or application associated with the message. Include the TRANSIENT attribute in the PL/I file declaration to specify access type. TRANSIENT indicates that the contents of the data set associated with the file are reestablished each time the data set is accessed. You can continually add records to the data set with one program while another program is running that continually removes records from the data set. Thus the data set can be considered to be a continuous first-in/first-out queue through which the records pass in transit between the message control program and the message processing program.

A data set associated with a TRANSIENT file differs from one associated with a DIRECT or SEQUENTIAL file in the following ways:

- Its contents are dynamic. Reading a record removes it from the data set.
- The ENDFILE condition is not defined for a TRANSIENT file. Instead, the PENDING condition is raised when the input queue is empty. This does not imply the queue will remain empty since records can be continually added.

In addition to TRANSIENT access, you can access a teleprocessing queue for input as a SEQUENTIAL file with consecutive organization (unless you use a READ statement option, such as EVENT, that is invalid for a TRANSIENT file). This support is provided by TCAM when it detects a request from a sequential access method (BSAM or QSAM). Your program is unaware of the fact that a TCAM queue is the source of input. You will not receive terminal identifiers in the character string referenced in the KEYTO option of the READ statement and the PENDING condition will not be raised. You can create a teleprocessing data set only by using a file with TRANSIENT access.

## Essential Information

To access a teleprocessing data set, the file name or value of the TITLE option on the OPEN statement must be the name of a DD statement that identifies the message queue in the QNAME parameter. For example:

```
//PLIFILE DD QNAME=process name
```

“process name” is the symbolic name of the TPROCESS macro, coded in your MCP, that defines the destination queue through which your messages will be routed. Your system programmer can provide the queue names to be used for your application.

For TRANSIENT OUTPUT files, the element expression you specify in the KEYFROM option must have as its value a terminal or program identifier known to your MCP. If you specify the TP(R) ENVIRONMENT option, indicating multiple-segment messages, you must indicate the position of record segments within a message, as described above.

---

## Defining Files for a Teleprocessing Data Set

You define a teleprocessing file with the attributes shown in the following declaration:

```
DCL filename FILE TRANSIENT RECORD
      INPUT | OUTPUT
      BUFFERED KEYED
      ENVIRONMENT(option-list);
```

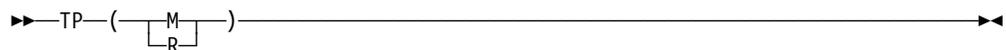
The file attributes are described in the *PL/I for MVS & VM Language Reference*. Required attributes and defaults are shown in Table 15 on page 111.

## Specifying ENVIRONMENT Options

For teleprocessing applications, the ENVIRONMENT options that you can specify are TP(M|R), RECSIZE(record-length), and BUFFERS(n).

### TP Option

Use TP to specify that the file is associated with a teleprocessing data set. A message can consist of one logical record or several logical records on the teleprocessing data set.



### TP(M)

specifies that each data transmission statement in your PL/I program transmits a complete message (which can be several logical records) to or from the data set.

### TP(R)

specifies that each data transmission statement in your PL/I program transmits a single logical record, which is a segment of a complete message.

One or more PL/I data transmission statements are required to completely transmit a message. On input, your PL/I application program must determine the end of a message by its own means; for example, this can be from

information embedded in the message. On output, your PL/I program must provide, for each logical record, its segment position within the message.

You indicate the position by a code in the first byte of the KEYFROM value, preceding the destination ID. The valid codes and their meanings are:

- |       |                                   |
|-------|-----------------------------------|
| 1     | First segment of a message        |
| blank | Intermediate segment of a message |
| 2     | Last segment in a message         |
| 3     | Only segment in a message.        |

Selection of TP(M) or TP(R) is dependent on the message format you specify in your MCP. Your system programmer can tell you which code to use.

### **RECSIZE Option**

Use the RECSIZE option to specify the size of the record variable (or input or output buffer, for locate mode) in your PL/I program. If you use the TP(M) option, this size should be equal to the length of all the logical records that constitute the message. If it is smaller, part of the message will be lost. If it is greater, the contents of the last part of the variable (or buffer) are undefined. If you specify the TP(R) option, this size must be the same as the logical record length.

You must specify RECSIZE.

### **BUFFERS Option**

Use the BUFFERS option to specify the number of intermediate buffers required to contain the longest message to be transmitted. The buffer size is defined in the message control program. If a message is too long for the buffers you specified, extra buffers must be obtained before processing can continue, which increases run time. The extra buffers are obtained by the operating system; you need not take any action.

---

## **Writing a TCAM Message Processing Program (TCAM MPP)**

You can access a TRANSIENT file with READ, WRITE, and LOCATE statements. You cannot use the EVENT option.

Use the READ statement for input, with either the INTO option or the SET option. You must give the KEYTO option. The origin name is assigned to the variable referenced in the KEYTO option. If the origin name is shorter than the character string referenced in the KEYTO option, it is padded on the right with blanks. If the KEYTO variable is a varying-length string, its current length is set to that of the origin name. The origin name should not be longer than the KEYTO variable (if it is, it is truncated), but in any case will not be longer than 8 characters. The data part of the message or record is assigned to the variable referenced in the INTO option, or the pointer variable referenced in the SET option is set to point to the data in the READ SET buffer.

A READ statement for the file will take the next message (or the next record from the current message) from the associated queue, assign the data part to the variable referenced in the READ INTO option (or set a pointer to point to the data in a READ SET buffer), and assign the character string origin identifier to the variable referenced in the KEYTO option. The PENDING condition is raised if the input queue is empty when a READ statement is executed.

You can use either the WRITE or the LOCATE statement for output. Either statement must have the KEYFROM option—for files declared with the TP(M) option, the first 8 characters of the value of the KEYFROM expression are used to identify the destination, which must be a recognized terminal or program identifier. For files declared with the TP(R) option, indicating multiple-segment messages, the first character of the value you specify in the KEYFROM expression must contain the message segment code as discussed above. The next 8 characters of the value are used to identify the destination. The data part of the message is transmitted from the variable referenced in the FROM option of the WRITE statement, or, in the case of LOCATE, a pointer variable is set to point to the location of the data in the output buffer.

The statements and options allowed for TRANSIENT files are given in Table 40. Some examples follow the figure.

*Table 40. Statements and Options Allowed for TRANSIENT Files*

<b>File declaration<sup>1</sup></b>	<b>Valid statements, with options you must include</b>	<b>Other options you can also include</b>
TRANSIENT INPUT	READ FILE(file-reference) INTO(reference) KEYTO(reference);  READ FILE(file-reference) SET(pointer-reference) KEYTO(reference);	
TRANSIENT OUTPUT	WRITE FILE(file-reference) FROM(reference) KEYFROM(expression);  LOCATE(based-variable) FILE(file-reference) KEYFROM(expression);	SET(pointer-reference)

**Notes:**

1. The complete file declaration would include the attributes FILE, RECORD, KEYED, BUFFERED, and the ENVIRONMENT attribute with either the TP(M) or the TP(R) option.

The following example illustrates the use of move mode in teleprocessing applications:

```

DECLARE (IN INPUT,OUT OUTPUT) FILE
        TRANSIENT ENV(TP(M) RECSIZE(124)),
        (INREC, OUTREC) CHARACTER(120)
        VARYING, TERM CHARACTER(8);

READ FILE(IN) INTO(INREC) KEYTO(TERM);
.
.
.
WRITE FILE(OUT) FROM(OUTREC)
        KEYFROM(TERM);

```

The files IN and OUT are given the attributes KEYED and BUFFERED because TRANSIENT implies these attributes. The TP(M) option indicates that a complete message will be transmitted. The input buffer for file IN contains the next message from the input queue.

The READ statement moves the message or record from the input buffer into the variable INREC. The character string identifying the origin is assigned to TERM. If the buffer is empty when the READ statement is executed (that is, if there are no messages in the queue), the PENDING condition is raised. The implicit action for the condition is described under “Handling PL/I Conditions.”

After processing, the message or record is held in OUTREC. The WRITE statement moves it to the output buffer, together with the value of TERM (which still contains the origin name unless another name has been assigned to it during processing). From the buffer, the message is transmitted to the correct queue for the destination, as specified by the value of TERM.

The next example is similar to the previous one, except that locate mode input is used.

```
DECLARE (IN INPUT,OUT OUTPUT) FILE
    TRANSIENT ENV(TP(M) RECSIZE(124)),
    MESSAGE CHARACTER(120) VARYING
    BASED(INPTR),
    TERM CHARACTER(8);

READ FILE(IN) SET(INPTR) KEYTO(TERM);
.
.
.
WRITE FILE(OUT) FROM(MESSAGE)
    KEYFROM(TERM);
```

The message data is processed in the input buffer, using the based variable MESSAGE, which has been declared with the pointer reference INPTR. (The variable MESSAGE will be aligned on a double word boundary.) The WRITE statement moves the processed data from the input to the output buffer; otherwise its effect is as described for the WRITE statement in the first example.

The technique used in this example would be useful in applications where the differences between processed and unprocessed messages were relatively simple, since the maximum size of input and output messages would be the same. If the length and structure of the output message could vary widely, depending on the text of the input message, locate mode output could be used to advantage. After the input message had been read in, a suitable based variable could be located in the output buffer (using the LOCATE statement), where further processing would take place. The message would be transmitted immediately before execution of the next WRITE or LOCATE statement for the file.

## Handling PL/I Conditions

The conditions that can be raised during teleprocessing transmission are TRANSMIT, KEY, RECORD, ERROR, and PENDING.

The TRANSMIT condition can be raised on input or output, as described for other types of transmission. In addition, for a TRANSIENT OUTPUT file, TRANSMIT can be raised in the following circumstances:

- The destination queue is full—TCAM rejected the message.
- For a file declared with the TP(R) ENVIRONMENT option, message segments were presented out of sequence.

The RECORD condition is raised under the same circumstances as for other types of transmission. The messages and records are treated as V-format records.

The ERROR condition is raised as for other types of transmission. It is also raised when the expression in the KEYFROM option is missing or invalid.

The KEY condition is raised if the expression in the KEYFROM option is syntactically valid but does not represent an origin or a destination name recognized by the MCP.

The PENDING condition is raised only during execution of a READ statement for a TRANSIENT file. When the PENDING condition is raised, the value returned by the ONKEY built-in function is a null string. The PL/I implicit action for the PENDING condition is as follows:

- If there is no ON-unit for the PENDING condition, the PL/I transmitter module waits for a message.
- If there is an ON-unit for the PENDING condition, and it executes a normal return, the transmitter waits for a message.
- If there is an ON-unit for the PENDING condition, and it does not return normally, the next execution of a READ statement again raises PENDING if no records have been added to the queue.

There is no PL/I condition associated with the occurrence of the last segment of a message. When you specify the TP(R) option, indicating multiple-segment messages, you are responsible for arranging the recognition of the end of the message.

## TCAM MPP Example

An example of a TCAM MPP and the job control language required to run it is shown in Figure 64 on page 262. The EXEC statement in the first part of the figure invokes the cataloged procedure IEL1CL to compile and link-edit the PL/I message processing program. The load module is stored in the library SYS1.MSGLIB under the member name MPPROC.

---

### Part 1. Compiling and link-editing the TCAM MPP

```
//JOBNAME JOB
// EXEC IELICL
//PLI.SYSIN DD *
MPPROC: PROC OPTIONS(MAIN);
  DCL INMSG FILE RECORD KEYED TRANSIENT ENV(TP(M) RECSIZE(100)),
  OUTMSG FILE RECORD KEYED TRANSIENT ENV(TP(M) RECSIZE(500)),
  INDATA CHAR(100),
  OUTDATA CHAR(500),
  TKEY CHAR(6);
  .
  .
  .
  OPEN FILE(INMSG) INPUT,FILE(OUTMSG) OUTPUT;
  .
  .
  .
  READ FILE(INMSG) KEYTO(TKEY) INTO(INDATA);
  .
  .
  .
  WRITE FILE(OUTMSG) KEYFROM(TKEY) FROM(OUTDATA);
  .
  .
  .
ENDTP: CLOSE FILE(INMSG),FILE(OUTMSG);
END MPPROC;
/*
//LKED.SYSLMOD DD DSNAME=SYS1.MSGLIB(MPPROC),DISP=OLD
```

### Part 2. Executing the TCAM MPP

```
//JOBNAME JOB ...
//JOBLIB DD DSNAME=SYS1.MSGLIB(MPPROC),DISP=SHR
// EXEC PGM=MPPROC
//INMSG DD QNAME=(INQUIRY)
//OUTMSG DD QNAME=(RESPONSE)
```

---

Figure 64. PL/I Message Processing Program

In the PL/I program, INMSG is declared as a teleprocessing file that can process messages up to 100 bytes long. Similarly, OUTMSG is declared as a teleprocessing file that can process messages up to 500 bytes long.

The READ statement gets a message from the queue. The terminal identifier, which is passed as a key by TCAM, is inserted into TKEY, the character string referenced in the KEYTO option. The record is placed in the INDATA variable for processing. The appropriate READ SET statement could also have been used here. The statements that process the data and place it in OUTDATA are omitted to simplify the example.

The WRITE statement moves the data from OUTDATA into the destination queue. The terminal identifier is taken from the character string in TKEY. An appropriate LOCATE statement could also have been used.

The TCAM MPP is executed in the second part of the example. The INMSG and OUTMSG DD statements associate the PL/I files TCAM MPP and OUTMSG with their respective main storage queues, that is, INQUIRY and RESPONSE.

---

## Part 4. Improving your program

<b>Chapter 13. Examining and Tuning Compiled Modules</b> . . . . .	265
Activating Hooks in Your Compiled Program Using IBMBHKS . . . . .	265
The IBMBHKS Programming Interface . . . . .	265
Obtaining Static Information about Compiled Modules Using IBMBSIR . . . . .	266
The IBMBSIR Programming Interface . . . . .	267
Obtaining Static Information as Hooks Are Executed Using IBMBHIR . . . . .	271
The IBMBHIR Programming Interface . . . . .	271
Examining Your Program's Run-Time Behavior . . . . .	272
Sample Facility 1: Examining Code Coverage . . . . .	272
Overall Setup . . . . .	272
Output Generated . . . . .	272
Source Code . . . . .	273
Sample Facility 2: Performing Function Tracing . . . . .	284
Overall Setup . . . . .	284
Output Generated . . . . .	284
Source Code . . . . .	284
Sample Facility 3: Analyzing CPU-Time Usage . . . . .	288
Overall Setup . . . . .	288
Output Generated . . . . .	288
Source Code . . . . .	290
<b>Chapter 14. Efficient Programming</b> . . . . .	305
Efficient Performance . . . . .	305
Tuning a PL/I Program . . . . .	305
Tuning a Program for a Virtual Storage System . . . . .	307
Global Optimization Features . . . . .	308
Expressions . . . . .	309
Common Expression Elimination . . . . .	309
Redundant Expression Elimination . . . . .	310
Simplification of Expressions . . . . .	310
Replacement of Constant Expressions . . . . .	311
Loops . . . . .	312
Transfer of Expressions from Loops . . . . .	312
Special Case Code for DO Statements . . . . .	313
Arrays and Structures . . . . .	313
Initialization of Arrays and Structures . . . . .	313
Structure and Array Assignments . . . . .	313
Elimination of Common Control Data . . . . .	313
In-Line Code . . . . .	314
In-line Code for Conversions . . . . .	314
In-line Code for Record I/O . . . . .	314
In-line Code for String Manipulation . . . . .	314
In-line Code for Built-In Functions . . . . .	314
Key handling for REGIONAL data sets . . . . .	314
REGIONAL(1) . . . . .	314
REGIONAL(2) and REGIONAL(3) . . . . .	314
Matching Format Lists with Data Lists . . . . .	315
Run-time Library Routines . . . . .	315
Use of Registers . . . . .	315
Program Constructs that Inhibit Optimization . . . . .	315

Global Optimization of Variables	316
ORDER and REORDER Options	316
ORDER Option	316
REORDER Option	316
Common Expression Elimination	318
Condition Handling for Programs with Common Expression Elimination	320
Transfer of Invariant Expressions	321
Redundant Expression Elimination	322
Other Optimization Features	322
Assignments and Initialization	323
Notes about Data Elements	323
Notes about Expressions and References	326
Notes about Data Conversion	329
Notes about Program Organization	331
Notes about Recognition of Names	332
Notes about Storage Control	332
Notes about Statements	334
Notes about Subroutines and Functions	338
Notes about Built-In Functions and Pseudovariables	338
Notes about Input and Output	339
Notes about Record-Oriented Data Transmission	340
Notes about Stream-Oriented Data Transmission	341
Notes about Picture Specification Characters	343
Notes about Condition Handling	344
Notes about multitasking	345

---

## Chapter 13. Examining and Tuning Compiled Modules

This chapter discusses how to obtain static information about your compiled program or other object modules of interest either during execution of your program or at any time. Specifically, it discusses:

- How to turn hooks on prior to execution by calling IBMBHKS (see “Activating Hooks in Your Compiled Program Using IBMBHKS”)
- How to call the Static Information Retrieval service IBMBSIR to retrieve static information about compiled modules (see “Obtaining Static Information about Compiled Modules Using IBMBSIR” on page 266)
- How to call the Hook Information Retrieval service IBMBHIR to obtain static information relative to hooks that are executed during your program's run (see “Obtaining Static Information as Hooks Are Executed Using IBMBHIR” on page 271)
- How to use IBMBHKS, IBMBSIR, and IBMBHIR via the hook exit in CEEBINT to examine your program's run-time behavior (see “Examining Your Program's Run-Time Behavior” on page 272).

These services are useful if you want to do any of the following:

- Examine and fine tune your program's run-time behavior by, for example, checking which statements, blocks, paths, labels, or calls are visited most
- Perform function tracing during your program's execution
- Examine CPU timing characteristics of your program's execution
- Find out information about any object module such as:
  - What options it was compiled with
  - Its size
  - The number and location of blocks in the module
  - The number and location of hooks in the module
  - The number and addresses of external entries in the module

These services are available in batch, PL/I multitasking, and CICS environments.

For information on how to establish the hook exit in CEEBINT, see the *Language Environment for MVS & VM Programming Guide*.

---

### Activating Hooks in Your Compiled Program Using IBMBHKS

The callable service IBMBHKS is provided to turn hooks on and off without the use of a debugging tool. It is available in batch, PL/I multitasking, and CICS environments.

#### The IBMBHKS Programming Interface

You can declare IBMBHKS in a PL/I program as follows:

```
DECLARE IBMBHKS EXTERNAL ENTRY( FIXED BIN(31,0), FIXED BIN(31,0) );
```

and invoke it with the following PL/I CALL statement:

```
CALL IBMBHKS( Function_code, Return_code );
```

The possible function codes are:

- 1 Turn on statement hooks
- 1 Turn off statement hooks
- 2 Turn on block entry hooks
- 2 Turn off block entry hooks
- 3 Turn on block exit hooks
- 3 Turn off block exit hooks
- 4 Turn on path hooks
- 4 Turn off path hooks
- 5 Turn on label hooks
- 5 Turn off label hooks
- 6 Turn on before-call hooks
- 6 Turn off before-call hooks
- 7 Turn on after-call hooks
- 7 Turn off after-call hooks.

The possible return codes are:

- 0 Successful
- 12 The debugging tool is active
- 16 Invalid function code passed.

**Note:** Turning on or off statement hooks or path hooks also turns on or off respectively block entry and block exit hooks. The reverse, however, is not true.

**Warning**

This service is meant to be used with the hook exit. It is an error to use IBMBHKS to turn on hooks when a hook exit has not been established in CEEBINT, and unpredictable results will occur in this case.

For examples of possible uses of IBMBHKS see "Examining Your Program's Run-Time Behavior" on page 272.

---

## Obtaining Static Information about Compiled Modules Using IBMBSIR

IBMBSIR is a Static Information Retrieval module that lets you determine static information about PL/I modules compiled with the TEST option. You can use it to interrogate static information about a compiled module (to find out, for example, what options it was compiled with), or you can use it recursively as part of a run-time monitoring process.

It is available in batch, PL/I multitasking, and CICS environments.

## The IBMBSIR Programming Interface

You invoke IBMBSIR with a PL/I CALL passing the address of a control block containing the following elements:

```
DECLARE
  1 SIR_DATA          BASED,
    2 SIR_FNCCODE     FIXED BIN(31), /* Function code          */
    2 SIR_RETCODE     FIXED BIN(31), /* Return code           */
    2 SIR_ENTRY       ENTRY,         /* Entry variable for module */
    2 SIR_MOD_DATA    POINTER,       /* Addr of module_data     */
    2 SIR_A_DATA      POINTER,       /* Addr of data for fnc code */
    2 SIR_END         CHAR(0);
```

where:

### SIR\_FNCCODE

specifies what type of information is desired, according to the following definitions:

- 1 Fill in the compile-time options information and the count of blocks in the module information control block (see the MODULE\_OPTIONS array and MODULE\_BLOCKS in Figure 65 on page 269).
- 2 Same function as 1 but also fill in the module's size, MODULE\_SIZE, in the module information control block.
- 3 Fill in all information specified in the module information control block; namely, compile-time options, count of blocks, module size, and counts of statements, paths, and external entries declared explicitly or implicitly.

Before invoking IBMBSIR with a function code of 4, you must have already invoked it with one of the first three function codes:

- 4 Fill in the information specified in the block information control block. The layout of this control block is given in Figure 66 on page 270.

BLOCK\_NAME\_LEN can be zero for unlabeled begin-blocks.

Before you invoke IBMBSIR with this function code, you must allocate the area for the control block and correctly set the fields BLOCK\_DATA and BLOCK\_COUNT.

Before invoking IBMBSIR with any of the following function codes, you must have already invoked it with a function code of 3:

- 5 Fill in the hook information block for all statement hooks. The layout of this control block is given in Figure 67 on page 270.

Note that statement hooks include block entry and exit hooks.

Before you invoke IBMBSIR with this function code, you must allocate the area for the control block and correctly set the fields HOOK\_DATA and HOOK\_COUNT.

HOOK\_IN\_LINE will be zero for all programs compiled with the STMT compile-time option, and for programs compiled with the NUMBER compile-time option, it will be nonzero only when a statement is one of a multiple in a source line.

HOOK\_OFFSET is always the offset of the hook from the primary entry point for the module, not the offset of the hook from the primary entry point for the block in which the hook occurs.

- 6 Fill in the hook information control block (see Figure 67 on page 270) for all path hooks.

Before you invoke IBMBSIR with this function code, you must allocate the area for the control block and correctly set the fields HOOK\_DATA and HOOK\_COUNT.

- 7 Fill in the external entry information control block. The layout of this control block is given in Figure 68 on page 270.

Before you invoke IBMBSIR with this function code, you must allocate the area for the control block and correctly set the fields EXTS\_DATA and EXTS\_COUNT.

EXTS\_EPA will give the entry point address for a module declared explicitly or implicitly in the program. It will be zero if the module has not been resolved at link-edit time.

**Note:** For all of the function codes you must also supply the SIR\_ENTRY and SIR\_MOD\_DATA parameters described below.

For function codes 4, 5, 6, and 7 you must supply the SIR\_A\_DATA parameter described below.

#### **SIR\_RETCODE**

is the return code:

- 0 Successful.
- 4 Module not compiled with appropriate TEST option.
- 8 Module not PL/I or not compiled with TEST option.
- 12 Invalid parameters passed.
- 16 Unknown function code.

#### **SIR\_ENTRY**

is the main entry point to your module.

#### **SIR\_MOD\_DATA**

is a pointer to the module information control block, shown in Figure 65 on page 269.

#### **SIR\_A\_DATA**

is a pointer to the block information control block, the hook information control block, or the external entries information control block, depending on which function code you are using.

The following figures show the layout of the control blocks:

Figure 66 on page 270 shows the block information control block.

Figure 67 on page 270 shows the hook information control block.

Figure 68 on page 270 shows the external entries information control block.

This field must be zero if you specify a function code of 1, 2, or 3. If you specify function codes 4, 5, 6, or 7, this field must point to the applicable control block:

<i>Function Code</i>	<i>Set Pointer to</i>
<b>4</b>	Block information control block
<b>5 or 6</b>	Hook information control block
<b>7</b>	External entries information control block.

---

```

DECLARE
1 MODULE_DATA      BASED,
2 MODULE_LEN      FIXED BIN(31),
2 MODULE_OPTIONS,
3 MODULE_GENERAL_OPTIONS,
4 MODULE_STMTNO  BIT(01),
4 MODULE_GONUM   BIT(01),
4 MODULE_CMPATV1 BIT(01),
4 MODULE_GRAPHIC BIT(01),
4 MODULE_OPT     BIT(01),
4 MODULE_INTER   BIT(01),
4 MODULE_GENIX   BIT(02),
3 MODULE_GENERAL_OPTIONS2,
4 MODULE_GEN2X   BIT(08),
3 MODULE_TEST_OPTIONS,
4 MODULE_TEST    BIT(01),
4 MODULE_STMT    BIT(01),
4 MODULE_PATH    BIT(01),
4 MODULE_BLOCK   BIT(01),
4 MODULE_TESTX   BIT(03),
4 MODULE_SYM     BIT(01),
3 MODULE_SYS_OPTIONS,
4 MODULE_CMS     BIT(01),
4 MODULE_CMSTP   BIT(01),
4 MODULE_MVS     BIT(01),
4 MODULE_TSO     BIT(01),
4 MODULE_CICS    BIT(01),
4 MODULE_IMS     BIT(01),
4 MODULE_SYSX    BIT(02),
2 MODULE_BLOCKS  FIXED BIN(31),
2 MODULE_SIZE    FIXED BIN(31),
2 MODULE_SHOOKS  FIXED BIN(31),
2 MODULE_PHOOKS  FIXED BIN(31),
2 MODULE_EXTS    FIXED BIN(31),
2 MODULE_DATA_END CHAR(0);

```

---

Figure 65. Module Information Control Block

---

```

DECLARE
1 BLOCK_TABLE    BASED,

2 BLOCK_DATA     POINTER,          /* Addr of BLOCK_INFO */
2 BLOCK_COUNT    FIXED BIN(31),    /* Count of blocks */

2 BLOCK_INFO( BLOCKS REFER(BLOCK_COUNT) ),

3 BLOCK_OFFSET  FIXED BIN(31),    /* Offset of block entry */
3 BLOCK_SIZE    FIXED BIN(31),    /* Size of block */
3 BLOCK_LEVEL   FIXED BIN(15),    /* Block nesting level */
3 BLOCK_PARENT  FIXED BIN(15),    /* Index for parent block */
3 BLOCK_CHILD   FIXED BIN(15),    /* Index for first child */
3 BLOCK_SIBLING FIXED BIN(15),    /* Index for next sibling */
3 BLOCK_NAME_LEN FIXED BIN(15),    /* Length of block name */
3 BLOCK_NAME_STR CHAR(34),        /* Block name */

2 BLOCK_TABLE_END CHAR(0);

```

---

*Figure 66. Block Information Control Block*

---

```

DECLARE
1 HOOK_TABLE     BASED,

2 HOOK_DATA      POINTER,          /* Addr of HOOK_INFO */
2 HOOK_COUNT     FIXED BIN(31),    /* Count of hooks */

2 HOOK_INFO( HOOKS REFER(HOOK_COUNT) ),

2 HOOK_OFFSET    FIXED BIN(31),    /* Offset of hook */
2 HOOK_NO        FIXED BIN(31),    /* Stmt number for hook */
2 HOOK_IN_LINE   FIXED BIN(15),    /* Stmt number within line */
2 HOOK_RESERVED  FIXED BIN(15),    /* Reserved */
2 HOOK_TYPE      FIXED BIN(15),    /* Hook type (=%PATHCODE) */
2 HOOK_BLOCK     FIXED BIN(15),    /* Block number for hook */

2 HOOK_TABLE_END CHAR(0);

```

---

*Figure 67. Hook Information Control Block*

---

```

DECLARE
1 EXTS_TABLE     BASED,

2 EXTS_DATA      POINTER,          /* Addr of EXTS_INFO */
2 EXTS_COUNT     FIXED BIN(31),    /* Count of entries */

2 EXTS_INFO( EXTS REFER(EXTS_COUNT) ),

2 EXTS_EPA       POINTER,          /* EPA for entry */

2 EXTS_TABLE_END CHAR(0);

```

---

*Figure 68. External Entries Information Control Block*

For examples of possible uses of IBMBSIR see “Examining Your Program's Run-Time Behavior” on page 272.

---

## Obtaining Static Information as Hooks Are Executed Using IBMBHIR

IBMBHIR is a Hook Information Retrieval module that lets you determine static information about hooks executed in modules compiled with the TEST option. It is available in batch, PL/I multitasking, and CICS environments.

### The IBMBHIR Programming Interface

You invoke IBMBHIR with a PL/I CALL passing, in order, the address of the control block shown below, the value of register 13 when the hook was executed, and the address of the hook that was executed. (These last two items are also the last two items passed to the hook exit.)

```
DECLARE
  1 HIR_DATA          BASED(HIR_PARMS),
  2 HIR_STG           FIXED BIN(31),    /* Size of this control block */
  2 HIR_EPA           POINTER,          /* Addr of module entry point */
  2 HIR_LANG_CODE     BIT(8),           /* Language code */
  2 HIR_PATH_CODE     BIT(8),           /* Path code for hook */
  2 HIR_NAME_LEN      FIXED BIN(15),    /* Length of module name */
  2 HIR_NAME_ADDR     POINTER,          /* Addr of module name */
  2 HIR_BLOCK         FIXED BIN(31),    /* Block count */
  2 HIR_END           CHAR(0);
```

These parameters, upon return from IBMBHIR, supply:

- Information about the module in which the hook was executed:

#### **HIR\_EPA**

the primary entry point address of the module (you could use this value as a parameter to IBMBSIR to obtain more data)

#### **HIR\_LANG\_CODE**

the programming language used to compile the module ('0A' BX representing PL/I, and '03' BX representing C)

#### **HIR\_NAME\_LEN**

the length of the name of the module

#### **HIR\_NAME\_ADDR**

the address of a nonvarying string containing the module name

- Information about the block in which the hook was executed:

#### **HIR\_BLOCK**

the block number for that block

- Information about the hook itself:

#### **HIR\_PATH\_CODE**

the %PATHCODE value associated with the hook.

The next section contains an example of one of the possible uses of IBMBHIR.

---

## Examining Your Program's Run-Time Behavior

This section shows some practical ways of using the services discussed in the first part of this chapter (IBMBHKS, IBMBSIR, and IBMHIR) via the hook exit in CEEBINT to monitor your program's run-time behavior. In particular, three sample facilities are presented, which demonstrate, respectively, how you can:

- Examine code coverage (see "Sample Facility 1: Examining Code Coverage")
- Perform function tracing (see "Sample Facility 2: Performing Function Tracing" on page 284)
- Analyze CPU-time usage (see page "Sample Facility 3: Analyzing CPU-Time Usage" on page 288).

For each facility, the overall setup is outlined briefly, the output is given, and the source code for the facility is shown.

### Sample Facility 1: Examining Code Coverage

The following example programs show how to establish a rather simple hook exit to report on code coverage.

#### Overall Setup

This facility consists of two programs: CEEBINT and HOOKUP. The CEEBINT module is coded so that:

- A hook exit to the HOOKUP program is established
- Calls to IBMBHKS are made to set hooks prior to execution.

At run time, whenever HOOKUP gains control (via the established hook exit), it calls IBMBSIR to obtain code coverage information on the MAIN procedure and those linked with it.

**Note:** The CEEBINT routine uses a recursive routine `Add_Module_to_List` to locate and save information on the MAIN module and all the modules linked with it. Before this routine is recursively invoked, a check should be made to see if the module to be added has already been added. If such a check is not made, the subroutine could call itself endlessly.

The SPROG suboption of the LANGLVL compile-time option is specified in order to enable the adding to a pointer that takes place in CEEBINT and the comparing of two pointers that takes place in HOOKUP.

#### Output Generated

The output given in Figure 69 on page 273 is generated during execution of a PL/I program called KNIGHT. The KNIGHT program's function is to determine the moves a knight must make to land on each square of a chess board only once.

The output is created by the HOOKUP program as employed in this facility.

---

Post processing

Data for block KNIGHT

Statement	Type	Visits	Percent
1	block entry	1	0.0264
14	before call	1	0.0264
14	after call	1	0.0264
21	start of do loop	63	1.6662
23	start of do loop	504	13.3298
25	if-true	229	6.0565
25	if-true	91	2.4067
29	if-false	138	3.6498
30	if-false	275	7.2732
32	if-true	63	1.6662
33	start of do loop	504	13.3298
34	if-true	224	5.9243
35	if-false	280	7.4054
42	if-false	0	0.0000
44	start of do loop	8	0.2115
65	block exit	1	0.0264

Data for block INITIALIZE\_RANKINGS

Statement	Type	Visits	Percent
48	block entry	1	0.0264
50	start of do loop	12	0.3173
51	start of do loop	144	3.8085
52	if-true	80	2.1158
53	if-false	64	1.6926
56	start of do loop	8	0.2115
57	start of do loop	64	1.6926
58	start of do loop	512	13.5413
59	if-true	176	4.6548
60	if-false	336	8.8865
64	block exit	1	0.0264

---

Figure 69. Code Coverage Produced by Sample Facility 1

## Source Code

The source code for Sample Facility 1 follows (CEEBINT in Figure 70, and HOOKUP in Figure 71 on page 280).

---

```

%PROCESS FLAG(1) GOSTMT STMT SOURCE;
%PROCESS OPT(2) TEST(NONE,NOSYM) LANGLVL(SPROG);
CEE Bint: Proc( Number, RetCode, RsnCode, FncCode, A_Main,
               UserWd, A_Exits )
               options(reentrant) reorder;

Dcl Number      fixed bin(31);      /* Number of args = 7      */
Dcl RetCode     fixed bin(31);      /* Return Code = 0        */
Dcl RsnCode     fixed bin(31);      /* Reason Code = 0        */
Dcl FncCode     fixed bin(31);      /* Function Code = 1      */
Dcl A_Main      pointer;            /* Address of Main Routine */
Dcl UserWd      fixed bin(31);      /* User Word               */
Dcl A_Exits     pointer;            /* A(Exits list)          */

Declare A_exit_list      pointer;

Declare
  1 Exit_list      based(A_exit_list),
  2 Exit_list_count fixed bin(31),
  2 Exit_list_slots,
  3 Exit_for_hooks pointer,
  2 Exit_list_end  char(0);

Declare
  1 Hook_exit_block based(Exit_for_hooks),
  2 Hook_exit_len   fixed bin(31),
  2 Hook_exit_rtn   pointer,
  2 Hook_exit_fnccode fixed bin(31),
  2 Hook_exit_retcode fixed bin(31),
  2 Hook_exit_rsncode fixed bin(31),
  2 Hook_exit_userword pointer,
  2 Hook_exit_ptr   pointer,
  2 Hook_exit_reserved pointer,
  2 Hook_exit_dsa   pointer,
  2 Hook_exit_addr  pointer,
  2 Hook_exit_end   char(0);

Declare
  1 Exit_area      based(Hook_exit_ptr),
  2 Exit_bdata     pointer,
  2 Exit_pdata     pointer,
  2 Exit_epa       pointer,
  2 Exit_mod_end   pointer,
  2 Exit_a_visits  pointer,
  2 Exit_prev_mod  pointer,
  2 Exit_area_end  char(0);

Declare (Addr,Entryaddr) builtin;
Declare (Stg,Sysnull)    builtin;

Declare IBMBHKS external entry( fixed bin(31), fixed bin(31));

Dcl HksFncStmt fixed bin(31) init(1) static;
Dcl HksFncEntry fixed bin(31) init(2) static;
Dcl HksFncExit fixed bin(31) init(3) static;
Dcl HksFncPath fixed bin(31) init(4) static;
Dcl HksFncLabel fixed bin(31) init(5) static;
Dcl HksFncBCall fixed bin(31) init(6) static;
Dcl HksFncACall fixed bin(31) init(7) static;
Dcl HksRetCode fixed bin(31);

```

---

Figure 70 (Part 1 of 6). Sample Facility 1: CEEBINT Module

```

/*****
/*
/* Following declares are used in setting up HOOKUP as the exit */
/*
/*****

Declare HOOKUP          external entry;

Declare IBMSIR external entry( pointer );

Declare
  1 Sir_data,
      2 Sir_fnccode    fixed bin(31), /* Function code          */
                                      /* 3: supply data for module */
                                      /* 4: supply data for blocks */
                                      /* 5: supply data for stmts */
                                      /* 6: supply data for paths */
                                      /*
      2 Sir_retcode    fixed bin(31), /* Return code            */
                                      /* 0: successful           */
                                      /* 4: not compiled with   */
                                      /* appropriate TEST opt. */
                                      /* 8: not PL/I or not    */
                                      /* compiled with TEST    */
                                      /* 12: unknown function code */
                                      /*
      2 Sir_entry      entry,          /* Entry var for module   */
                                      /*
      2 Sir_mod_data   pointer,        /* A(module_data)        */
                                      /*
      2 Sir_a_data     pointer,        /* A(data for function code) */
                                      /*
      2 Sir_end        char(0);        /*

Declare
  1 Module_data,
      2 Module_len     fixed bin(31), /* = STG(Module_data)    */
                                      /*
      2 Module_options, /* Compile time options  */
                                      /*
      3 Module_general_options, /*
      4 Module_stmtno  BIT(01), /* Stmt number table does */
                                      /* not exist               */
      4 Module_gonum   BIT(01), /* Table has GONUMBER format */
      4 Module_cmpatv1 BIT(01), /* compiled with CMPAT(V1) */
      4 Module_graphic BIT(01), /* compiled with GRAPHIC */
      4 Module_opt     BIT(01), /* compiled with OPTIMIZE */
      4 Module_inter   BIT(01), /* compiled with INTERRUPT */
      4 Module_gen1x   BIT(02), /* Reserved                */
                                      /*
      3 Module_general_options2, /*
      4 Module_gen2x   BIT(08), /* Reserved                */
                                      /*
      3 Module_test_options, /*
      4 Module_test    BIT(01), /* compiled with TEST      */
      4 Module_stmt    BIT(01), /* STMT suboption is valid */
      4 Module_path    BIT(01), /* PATH suboption is valid */
      4 Module_block   BIT(01), /* BLOCK suboption is valid */
      4 Module_testx   BIT(03), /* Reserved                */
      4 Module_sym     BIT(01), /* SYM suboption is valid */
                                      /*

```

Figure 70 (Part 2 of 6). Sample Facility 1: CEEBINT Module

---

```

3 Module_sys_options,      /* */
4 Module_cms      BIT(01), /* SYSTEM(CMS) */
4 Module_cmstp    BIT(01), /* SYSTEM(CMSTP) */
4 Module_mvs      BIT(01), /* SYSTEM(MVS) */
4 Module_tso      BIT(01), /* SYSTEM(TSO) */
4 Module_cics     BIT(01), /* SYSTEM(CICS) */
4 Module_ims      BIT(01), /* SYSTEM(IMS) */
4 Module_sysx     BIT(02), /* Reserved */
                          /* */
2 Module_blocks   fixed bin(31), /* Count of blocks */
                          /* */
2 Module_size     fixed bin(31), /* Size of module */
                          /* */
2 Module_shooks   fixed bin(31), /* Count of stmt hooks */
                          /* */
2 Module_phooks   fixed bin(31), /* Count of path hooks */
                          /* */
2 Module_exts     fixed bin(31), /* Count of external entries */
                          /* */
2 Module_data_end char(0);

Declare
1 Block_table      based(A_block_table),
2 Block_a_data     pointer,
2 Block_count      fixed bin(31),
2 Block_data(Block refer(Block_count)),
3 Block_offset     fixed bin(31),
3 Block_size       fixed bin(31),
3 Block_level      fixed bin(15),
3 Block_parent     fixed bin(15),
3 Block_child      fixed bin(15),
3 Block_sibling    fixed bin(15),
3 Block_name       char(34) varying,
2 Block_table_end char(0);

Declare
1 Stmt_table       based(A_stmt_table),
2 Stmt_a_data      pointer,
2 Stmt_count       fixed bin(31),
2 Stmt_data(Stmts refer(Stmt_count)),
3 Stmt_offset      fixed bin(31),
3 Stmt_no          fixed bin(31),
3 Stmt_lineno      fixed bin(15),
3 Stmt_reserved    fixed bin(15),
3 Stmt_type        fixed bin(15),
3 Stmt_block       fixed bin(15),
2 Stmt_table_end   char(0);

Declare
1 Path_table       based(A_path_table),
2 Path_a_data      pointer,
2 Path_count       fixed bin(31),
2 Path_data(Paths refer(Path_count)),
3 Path_offset      fixed bin(31),
3 Path_no          fixed bin(31),
3 Path_lineno      fixed bin(15),
3 Path_reserved    fixed bin(15),
3 Path_type        fixed bin(15),
3 Path_block       fixed bin(15),
2 Path_table_end   char(0);

Declare Blocks      fixed bin(31);
Declare Stmts       fixed bin(31);
Declare Paths       fixed bin(31);
Declare Visits      fixed bin(31);

```

---

Figure 70 (Part 3 of 6). Sample Facility 1: CEEBINT Module

---

```

Declare A_block_table    pointer;
Declare A_path_table     pointer;
Declare A_stmt_table     pointer;

Declare
  1 Hook_table          based(Exit_a_visits),
  2 Hook_data_size      fixed bin(31),
  2 Hook_data(Visits refer(Hook_data_size)),
  3 Hook_visits         fixed bin(31),
  2 Hook_data_end       char(0);

Declare A_visits         pointer;
Declare A_type           pointer;
Declare Previous_in_chain pointer;

/*****
/*
/* Following code is used to set up the hook exit control block */
/*
*****/

Allocate Exit_list;
Exit_list_count = 1;

A_Exits = A_exit_list;

Allocate Hook_exit_block;
Hook_exit_len = Stg(Hook_Exit_block);

/*****
/*
/* Following code sets up HOOKUP as the hook exit */
/*
*****/

Hook_exit_rtn = Entryaddr(HOOKUP);

Previous_in_chain = Sysnull();

Call Add_module_to_list( A_main );

Call IBMBHKS( HksFncEntry, HksRetCode );
Call IBMBHKS( HksFncExit, HksRetCode );
Call IBMBHKS( HksFncPath, HksRetCode );

```

---

Figure 70 (Part 4 of 6). Sample Facility 1: CEEBINT Module

---

```

/*****
/*
/* Following subroutine retrieves all the static information
/* available on the MAIN routine and those linked with it
/*
/*
/*****

Add_module_to_list: Proc( In_epa ) recursive;

Dcl In_epa      pointer;
Dcl Next_epa   pointer;
Dcl Inx        fixed bin(31);

Declare
  1 Exts_table      based(A_exts_table),
  2 Exts_a_data     pointer,
  2 Exts_count      fixed bin(31),
  2 Exts_data(Exts refer(Exts_count)),
  3 Exts_epa        pointer,
  2 Exts_table_end  char(0);

Declare Exts      fixed bin(31);
Declare A_exts_table pointer;

Sir_fnccode = 3;

Entryaddr(Sir_entry) = In_epa;
Sir_mod_data = Addr(Module_data);

Module_len = Stg(Module_data);

Call IBMSIR(Addr(Sir_data));

If ( Sir_retcde = 0 )
& ( Module_path ) then
Do;
  Sir_fnccode = 4;

  Blocks = Module_blocks;
  Allocate Block_table;
  Block_a_data = ADDR(Block_data);
  Sir_a_data = A_block_table;

  Call IBMSIR(Addr(Sir_data));

  Sir_fnccode = 6;

  Paths = Module_phooks;
  Allocate Path_table;
  Path_a_data = Addr(Path_data);
  Sir_a_data = A_path_table;

  Call IBMSIR(Addr(Sir_data));

  /* Allocate areas needed          */

  Allocate Exit_area;

  Exit_prev_mod = Previous_in_chain;
  Previous_in_chain = Hook_exit_ptr;

  Exit_pdata = A_path_table;
  Exit_bdata = A_block_table;

  Exit_epa = In_epa;
  Exit_mod_end = Exit_epa + module_size;

```

---

Figure 70 (Part 5 of 6). Sample Facility 1: CEEBINT Module

---

```

Visits = Paths;
Allocate Hook_table Set(Exit_a_visits);
Hook_visits = 0;

If Module_exts = 0 then;
Else
  Do;
    Sir_fnccode = 7;

    Exts = Module_exts;
    Allocate Exts_table;
    Exts_a_data = Addr(Exts_data);
    Sir_a_data = A_exts_table;

    Call IBMBSIR(Addr(Sir_data));

    If Sir_retcode = 0 then
      Do;
        Do Inx = 1 to Exts;
          Next_epa = A_exts_table->Exts_epa(Inx);
          Call Add_module_to_list( Next_epa );
        End;
        Free Exts_table;
      End;
    Else;
  End;
End;
Else;

End Add_module_to_list;

End;

```

---

*Figure 70 (Part 6 of 6). Sample Facility 1: CEEBINT Module*

---

```

%PROCESS FLAG(1) GOSTMT STMT SOURCE;
%PROCESS OPT(2) TEST(NONE,NOSYM) LANGLVL(SPROG);
HOOKUP: Proc( Exit_for_hooks ) reorder;

    Dcl Exit_for_hooks    pointer;    /* Address of exit list    */

    Declare
    1 Hook_exit_block    based(Exit_for_hooks),
      2 Hook_exit_len    fixed bin(31),
      2 Hook_exit_rtn    pointer,
      2 Hook_exit_fnccode fixed bin(31),
      2 Hook_exit_retcode fixed bin(31),
      2 Hook_exit_rsncode fixed bin(31),
      2 Hook_exit_userword pointer,
      2 Hook_exit_ptr    pointer,
      2 Hook_exit_reserved pointer,
      2 Hook_exit_dsa    pointer,
      2 Hook_exit_addr    pointer,
      2 Hook_exit_end    char(0);

    Declare
    1 Exit_area          based(Module_data),
      2 Exit_bdata       pointer,
      2 Exit_pdata       pointer,
      2 Exit_epa         pointer,
      2 Exit_last        pointer,
      2 Exit_a_visits    pointer,
      2 Exit_prev_mod    pointer,
      2 Exit_area_end    char(0);

    Declare
    1 Path_table         based(Exit_pdata),
      2 Path_a_data      pointer,
      2 Path_count       fixed bin(31),
      2 Path_data(32767),
        3 Path_offset    fixed bin(31),
        3 Path_no        fixed bin(31),
        3 Path_lineno    fixed bin(15),
        3 Path_reserved  fixed bin(15),
        3 Path_type      fixed bin(15),
        3 Path_block     fixed bin(15),
      2 Path_table_end   char(0);

    Declare
    1 Block_table        based(Exit_bdata),
      2 Block_a_data     pointer,
      2 Block_count      fixed bin(31),
      2 Block_data(32767),
        3 Block_offset   fixed bin(31),
        3 Block_size     fixed bin(31),
        3 Block_level    fixed bin(15),
        3 Block_parent   fixed bin(15),
        3 Block_child    fixed bin(15),
        3 Block_sibling  fixed bin(15),
        3 Block_name     char(34) varying,
      2 Block_table_end  char(0);

    Declare
    1 Hook_table         based(Exit_a_visits),
      2 Hook_data_size   fixed bin(31),
      2 Hook_data(32767),
        3 Hook_visits    fixed bin(31),
      2 Hook_data_end    char(0);

```

---

Figure 71 (Part 1 of 4). Sample Facility 1: HOOKUP Program

---

```

Declare Ps          fixed bin(31);
Declare Ix          fixed bin(31);
Declare Jx          fixed bin(31);
Declare Total      float dec(06);
Declare Percent    Fixed dec(6,4);
Declare Col1       char(33);
Declare Col2       char(14);
Declare Col3       char(10);

Declare Sysnull    Builtin;

Declare Module_data pointer;

Module_data = Hook_exit_ptr;

/*****
*/
/* Search for hook address in chain of modules */
/*
*/
*****/

Do While ( Hook_exit_addr < Exit_epa | Hook_exit_addr > Exit_last )
  Until ( Module_data = Sysnull() );
  Module_data = Exit_prev_mod;
End;

/*****
*/
/* If not, found */
/* IBMHIR could be called to find address of entry point */
/* for the module */
/* IBMBSIR could then be called as in CEEBINT to add */
/* module to the chain of known modules */
/*
*/
*****/

If Module_data = Sysnull() then;
Else
  Do;
    Ps = Hook_exit_addr - Exit_epa;

    /*****
    */
    /* A binary search could be done here and such a search */
    /* would be much more efficient for large programs */
    /*
    */
    *****/

    Do Ix = 1 to Path_count
      While ( Ps /= Path_offset(Ix) );
    End;

    If (Ix > 0) & (Ix <= Path_count) then
      Hook_visits(Ix) = Hook_visits(Ix) + 1;
    Else;

```

---

Figure 71 (Part 2 of 4). Sample Facility 1: HOOKUP Program

---

```

/*****
/*
/* If hook type is for block exit
/* AND
/* block being exited is the first block in a procedure
/* AND
/* that procedure is the MAIN procedure, then
/* invoke the post processing routine
/*
/* Note that these conditions might never be met, for
/* example, if SIGNAL FINISH were issued or if an
/* EXEC CICS RETURN were issued
/*
*****/

If Path_type(Ix) = 2
& Path_block(Ix) = 1
& Exit_prev_mod = Sysnull() then
Do;
Put skip list ( ' ' );
Put skip list ( ' ' );

Put skip list ( 'Post processing' );

Module_data = Hook_exit_ptr;

Do Until ( Module_data = Sysnull() );
Call Report_data;
Module_data = Exit_prev_mod;
End;

End;
Else;
End;

Hook_exit_retcode = 4;
Hook_exit_rsnocode = 0;

Report_data: Proc;

Total = 0;
Do Jx = 1 to Path_count;
Total = Total + Hook_visits(Jx);
End;

Put skip list ( ' ' );
Do Jx = 1 to Path_count;
If Path_type(Jx) = 1 then
do;
Put skip list ( ' ' );
Put skip list ( 'Data for block ' ||
Block_name(Path_block(Jx)) );
Put skip list ( ' ' );
Col1 = ' Statement Type';
Col2 = ' Visits';
Col3 = ' Percent';
Put skip list ( Col1 || ' ' || Col2 || ' ' || Col3 );
Put skip list ( ' ' );
end;
Else;

```

---

Figure 71 (Part 3 of 4). Sample Facility 1: HOOKUP Program

---

```

Select ( Path_type(Jx) );
  When ( 1 )
    Col1 = Path_no(Jx) || ' block entry';
  When ( 2 )
    Col1 = Path_no(Jx) || ' block exit';
  When ( 3 )
    Col1 = Path_no(Jx) || ' label';
  When ( 4 )
    Col1 = Path_no(Jx) || ' before call';
  When ( 5 )
    Col1 = Path_no(Jx) || ' after call';
  When ( 6 )
    Col1 = Path_no(Jx) || ' start of do loop';
  When ( 7 )
    Col1 = Path_no(Jx) || ' if-true';
  When ( 8 )
    Col1 = Path_no(Jx) || ' if-false';
  Otherwise
    Col1 = Path_no(Jx);
End;

Col2 = Hook_visits(Jx);
Percent = 100 * (Hook_visits(Jx)/Total);
Put skip list ( Col1 || ' ' || Col2 || ' ' || Percent );
End;

Put skip list ( ' ' );
Put skip list ( ' ' );
Put skip list ( ' ' );

End Report_data;

End;

```

---

*Figure 71 (Part 4 of 4). Sample Facility 1: HOOKUP Program*

## Sample Facility 2: Performing Function Tracing

The following example programs show how to establish a rather simple hook exit to perform function tracing.

### Overall Setup

This facility consists of two programs: CEEBINT and HOOKUPT. The CEEBINT module is coded such that:

- A hook exit to the HOOKUPT program is established
- Calls to IBMBHKS are made to set hooks prior to execution.

At run time, whenever HOOKUPT gains control (via the established hook exit), it calls IBMBHIR to obtain information to create a function trace.

### Output Generated

The output given in Figure 72 below is generated during execution of a PL/I program called KNIGHT. The KNIGHT program's function is to determine the moves a knight must make to land on each square of a chess board only once.

The output is created by the HOOKUPT program as employed in this facility.

---

```
Entry hook in KNIGHT
Exit hook in KNIGHT
```

---

*Figure 72. Function Trace Produced by Sample Facility 2*

**Note:** In a more complicated program, many more entry and exit messages would be produced.

### Source Code

The source code for Sample Facility 2 follows (CEEBINT in Figure 73, and HOOKUPT in Figure 74 on page 287).

---

```

%PROCESS FLAG(1) GOSTMT STMT SOURCE;
%PROCESS OPT(2) TEST(NONE,NOSYM) LANGLVL(SPROG);
CEE Bint: Proc( Number, RetCode, RsnCode, FncCode, A_Main,
               UserWd, A_Exits )
               options(reentrant) reorder;

Dcl Number      fixed bin(31);      /* Number of args = 7      */
Dcl RetCode     fixed bin(31);      /* Return Code = 0        */
Dcl RsnCode     fixed bin(31);      /* Reason Code = 0        */
Dcl FncCode     fixed bin(31);      /* Function Code = 1       */
Dcl A_Main      pointer;            /* Address of Main Routine */
Dcl UserWd      fixed bin(31);      /* User Word               */
Dcl A_Exits     pointer;            /* A(Exits list)          */

Declare A_exit_list      pointer;

Declare
  1 Exit_list      based(A_exit_list),
  2 Exit_list_count fixed bin(31),
  2 Exit_list_slots,
  3 Exit_for_hooks pointer,
  2 Exit_list_end  char(0);

Declare
  1 Hook_exit_block based(Exit_for_hooks),
  2 Hook_exit_len   fixed bin(31),
  2 Hook_exit_rtn   pointer,
  2 Hook_exit_fnccode fixed bin(31),
  2 Hook_exit_retcode fixed bin(31),
  2 Hook_exit_rsncode fixed bin(31),
  2 Hook_exit_userword pointer,
  2 Hook_exit_ptr   pointer,
  2 Hook_exit_reserved pointer,
  2 Hook_exit_dsa   pointer,
  2 Hook_exit_addr  pointer,
  2 Hook_exit_end   char(0);

Declare
  1 Exit_area      based(Hook_exit_ptr),
  2 Exit_bdata     pointer,
  2 Exit_pdata     pointer,
  2 Exit_epa       pointer,
  2 Exit_mod_end   pointer,
  2 Exit_a_visits  pointer,
  2 Exit_prev_mod  pointer,
  2 Exit_area_end  char(0);

Declare (Addr,Entryaddr) builtin;
Declare (Stg,Sysnull)    builtin;

Declare IBMBHKS external entry( fixed bin(31), fixed bin(31));

Dcl HksFncStmt fixed bin(31) init(1) static;
Dcl HksFncEntry fixed bin(31) init(2) static;
Dcl HksFncExit fixed bin(31) init(3) static;
Dcl HksFncPath fixed bin(31) init(4) static;
Dcl HksFncLabel fixed bin(31) init(5) static;
Dcl HksFncBCall fixed bin(31) init(6) static;
Dcl HksFncACall fixed bin(31) init(7) static;
Dcl HksRetCode fixed bin(31);

```

---

Figure 73 (Part 1 of 2). Sample Facility 2: CEEBINT Module

---

```

/*****
/*
/* Following code is used to set up the hook exit control block */
/*
/*****

Allocate Exit_list;
Exit_list_count = 1;

A_Exits = A_exit_list;

Allocate Hook_exit_block;
Hook_exit_len = Stg(Hook_Exit_block);

/*****
/*
/* Following code sets up HOOKUPT as the hook exit */
/*
/*****

Declare HOOKUPT external entry;

Hook_exit_rtn = Entryaddr(HOOKUPT);

Call IBMBHKS( HksFncEntry, HksRetCode );
Call IBMBHKS( HksFncExit, HksRetCode );

End;

```

---

*Figure 73 (Part 2 of 2). Sample Facility 2: CEEBINT Module*

---

```

%PROCESS FLAG(1) GOSTMT STMT SOURCE;
%PROCESS OPT(2) TEST(NONE,NOSYM) LANGLVL(SPROG);
HOOKUPT: Proc( Exit_for_hooks ) reorder;

    Decl Exit_for_hooks    pointer;    /* Address of exit list    */

    Declare
    1 Hook_exit_block    based(Exit_for_hooks),
    2 Hook_exit_len      fixed bin(31),
    2 Hook_exit_rtn      pointer,
    2 Hook_exit_fnccode  fixed bin(31),
    2 Hook_exit_retcode  fixed bin(31),
    2 Hook_exit_rsncode  fixed bin(31),
    2 Hook_exit_userword pointer,
    2 Hook_exit_ptr      pointer,
    2 Hook_exit_reserved pointer,
    2 Hook_exit_dsa      pointer,
    2 Hook_exit_addr     pointer,
    2 Hook_exit_end      char(0);

    Declare
    1 Hook_data,
    2 Hook_stg           fixed bin(31),
    2 Hook_epa           pointer,
    2 Hook_lang_code     aligned bit(8),
    2 Hook_path_code     aligned bit(8),
    2 Hook_name_len      fixed bin(15),
    2 Hook_name_addr     pointer,
    2 Hook_block_count   fixed bin(31),
    2 Hook_reserved      fixed bin(31),
    2 Hook_data_end      char(0);

    Declare IBMBHIR      external entry;
    Declare Chars        char(256) based;
    Declare (Addr,Substr) builtin;

    Call IBMBHIR( Addr(Hook_data), Hook_exit_dsa, Hook_exit_addr );

    If Hook_block_count = 1 then
    Select ( Hook_path_code );
    When ( 1 )
    Put skip list( 'Entry hook in ' ||
    Substr(Hook_name_addr->Chars,1,Hook_name_len) );
    When ( 2 )
    Put skip list( 'Exit hook in ' ||
    Substr(Hook_name_addr->Chars,1,Hook_name_len) );
    Otherwise
    ;
    End;
    Else;

    End;

```

---

Figure 74. Sample Facility 2: HOOKUPT Program

## Sample Facility 3: Analyzing CPU-Time Usage

This facility extends the code-coverage function of Sample Facility 1 to also report on CPU-time usage.

### Overall Setup

This facility consists of four programs: CEEBINT, HOOKUP, TIMINI, and TIMCPU. The CEEBINT module is coded so that:

- A hook exit to the HOOKUP program is established
- Calls to IBMBHKS are made to set hooks prior to execution.

At run time, whenever HOOKUP gains control (via the established hook exit), it calls IBMBSIR to obtain code coverage information on the MAIN procedure and those linked with it. This is identical to the function of Sample Facility 1.

In addition, this HOOKUP program makes calls to the assembler routines TIMINI and TIMCPU to obtain information on CPU-time usage.

The SPROG suboption of the LANGLVL compile-time option is specified in order to enable the adding to a pointer that takes place in CEEBINT and the comparing of two pointers that takes place in HOOKUP.

### Output Generated

The output given in Figure 75 on page 289 is generated during execution of a sample program named EXP98 under CMS or MVS. The main procedure was compiled with the TEST(ALL) option.

The output is created by the HOOKUP program as employed in this facility.

---

Data for block EXP98

Statement Type	--- Visits ---		--- CPU Time ---	
	Number	Percent	Milliseconds	Percent
1 block entry	1	0.0201		
16 start of do loop	24	0.4832	83.768	0.6036
26 start of do loop	38	0.7652	107.254	0.7729
27 start of do loop	38	0.7652	102.159	0.7362
28 if true	38	0.7652	103.066	0.7427
30 start of do loop	456	9.1824	1233.827	8.8915
31 before call	456	9.1824	1237.831	8.9203
31 after call	456	9.1824	1238.400	8.9244
41 before call	38	0.7652	102.491	0.7385
41 after call	38	0.7652	103.038	0.7425
49 start of do loop	38	0.7652	102.029	0.7352
50 if true	0	0.0000	0.000	0.0000
51 if true	0	0.0000	0.000	0.0000
52 if true	0	0.0000	0.000	0.0000
54 start of do loop	304	6.1216	827.971	5.9667
55 if true	76	1.5304	206.831	1.4905
56 if true	0	0.0000	0.000	0.0000
57 if true	38	0.7652	104.351	0.7520
61 block exit	1	0.0201	2.703	0.0194
Totals for block	2040	41.0790	5555.719	40.0364

Data for block V1B

Statement Type	--- Visits ---		--- CPU Time ---	
	Number	Percent	Milliseconds	Percent
32 block entry	456	9.1824		
33 start of do loop	456	9.1824	1251.487	9.0187
34 if true	456	9.1824	1251.125	9.0161
35 if true	456	9.1824	1501.528	10.8206
36 if true	0	0.0000	0.000	0.0000
37 if true	456	9.1824	1275.072	9.1887
39 block exit	456	9.1824	1256.781	9.0569
Totals for block	2736	55.0944	6535.993	47.1010

Data for block V1C

Statement Type	--- Visits ---		--- CPU Time ---	
	Number	Percent	Milliseconds	Percent
42 block entry	38	0.7652		
43 start of do loop	38	0.7652	105.529	0.7604
44 if true	38	0.7652	106.233	0.7655
45 if true	0	0.0000	0.000	0.0000
46 if true	38	0.7652	106.481	0.7673
48 block exit	38	0.7652	104.547	0.7534
Totals for block	190	3.8260	422.790	3.0466

---

Figure 75. CPU-Time Usage and Code Coverage Reported by Sample Facility 3

The performance of this facility depends on the number of hook exits invoked. Collecting the data above increased the CPU time of EXP98 by approximately forty-four times. Each CPU-time measurement indicates the amount of virtual CPU time that was used since the previous hook was executed. Note that the previous hook is not necessarily the previous hook in the figure.

In the above data, the percent columns for the number of visits and the CPU time are very similar. This will not always be the case, especially where hidden code (like library calls or supervisor services) is involved.

## Source Code

The source code for Sample Facility 3 is shown in the following figures:

CEEBINT            Figure 76  
HOOKUP            Figure 77 on page 297  
TIMINI (MVS)      Figure 78 on page 302  
TIMINI (CMS)      Figure 79 on page 302  
TIMCPU (MVS)      Figure 80 on page 303  
TIMCPU (CMS)      Figure 81 on page 304.

---

```
%PROCESS TEST(NONE,NOSYM) LANGLVL(SPROG);
CEEBINT: PROC( Number, RetCode, RsnCode, FncCode, A_Main,
              UserWd, A_Exits )
              OPTIONS(REENTRANT) REORDER;

DCL Number      FIXED BIN(31);        /* Number of args = 7        */
DCL RetCode     FIXED BIN(31);        /* Return Code = 0          */
DCL RsnCode     FIXED BIN(31);        /* Reason Code = 0          */
DCL FncCode     FIXED BIN(31);        /* Function Code = 1        */
DCL A_Main      POINTER;              /* Address of Main Routine  */
DCL UserWd      FIXED BIN(31);        /* User Word                */
DCL A_Exits     POINTER;              /* A(Exits list)            */

/*****
/* This routine gets invoked at initialization of the MAIN        */
/* Structures and variables for use in establishing a hook exit  */
/*                                                                    */
/*****

DECLARE A_exit_list      POINTER;
DECLARE Based_ptr        POINTER BASED;
DECLARE Entry_var        ENTRY VARIABLE;

DECLARE Entryaddr        BUILTIN;
DECLARE (Stg, Sysnull)   BUILTIN;

DECLARE
  1 Exit_list            BASED(A_exit_list),
  2 Exit_list_count     FIXED BIN(31),
  2 Exit_list_slots,
  3 Exit_for_hooks      POINTER,
  2 Exit_list_end        CHAR(0);

DECLARE
  1 Hook_exit_block     BASED(Exit_for_hooks),
  2 Hook_exit_len        FIXED BIN(31),
  2 Hook_exit_rtn        POINTER,
  2 Hook_exit_fnccode    FIXED BIN(31),
  2 Hook_exit_retcode    FIXED BIN(31),
  2 Hook_exit_rsncode    FIXED BIN(31),
  2 Hook_exit_userword   FIXED BIN(31),
  2 Hook_exit_ptr        POINTER,
  2 Hook_exit_reserved   POINTER,
  2 Hook_exit_dsa        POINTER,
  2 Hook_exit_addr        POINTER,
  2 Hook_exit_end        CHAR(0);

DECLARE
  1 Exit_area            BASED(Hook_exit_ptr),
  2 Exit_bdata            POINTER,
  2 Exit_pdata            POINTER,
  2 Exit_epa              POINTER,
  2 Exit_xtra             POINTER,
  2 Exit_area_end        CHAR(0);
```

---

Figure 76 (Part 1 of 7). Sample Facility 3: CEEBINT Module

---

```

DECLARE
  1 Hook_table      BASED(Exit_xtra),
  2 Hook_data_size  FIXED BIN(31),
  2 Hook_data(Visits REFER(Hook_data_size)),
  3 Hook_visits     FIXED BIN(31),
  2 Hook_data_end   CHAR(0);

/* the name of the routine that gets control at hook exit */

DECLARE HOOKUP      EXTERNAL ENTRY ( POINTER );

/*****
/*
/* End of structures and variables for use with hook exit
/*
/*
*****/

/*****
/*
/* Structures and variables for use in invoking IBMBSIR
/*
/*
*****/

DECLARE
  1 Sir_data,
      /*
      /* Function code
      /* 1: supply data for module
      /* 2: supply data for blocks
      /* 3: supply data for stmts
      /* 4: supply data for paths
      /*
      /* Return code
      /* 0: successful
      /* 4: not compiled with
      /* appropriate TEST opt.
      /* 8: not PL/I or not
      /* compiled with TEST
      /* 12: unknown function code
      /*
      /* Entry var for module
      /*
      /* A(module_data)
      /*
      /* A(data for function code)
      /*
      /*
      /*
      /* CHAR(0);
      /*
      /*

DECLARE
  1 Module_data,
      /*
      /* = STG(Module_data)
      /*
      /* Compile time options
      /*
      /*
      /*
      /* Stmt number table does
      /* not exist
      /*

```

---

Figure 76 (Part 2 of 7). Sample Facility 3: CEEBINT Module

---

```

4 Module_gonum BIT(01), /* Table has GONUMBER format */
4 Module_cmpatv1 BIT(01), /* compiled with CMPAT(V1) */
4 Module_graphic BIT(01), /* compiled with GRAPHIC */
4 Module_opt BIT(01), /* compiled with OPTIMIZE */
4 Module_inter BIT(01), /* compiled with INTERRUPT */
4 Module_gen1x BIT(02), /* Reserved */
/*
3 Module_general_options2, /*
/*
4 Module_gen2x BIT(08), /* Reserved */
/*
3 Module_test_options, /*
4 Module_test BIT(01), /* compiled with TEST */
4 Module_stmt BIT(01), /* STMT suboption is valid */
4 Module_path BIT(01), /* PATH suboption is valid */
4 Module_block BIT(01), /* BLOCK suboption is valid */
4 Module_testx BIT(03), /* Reserved */
4 Module_sym BIT(01), /* SYM suboption is valid */
/*
3 Module_sys_options, /*
4 Module_cms BIT(01), /* SYSTEM(CMS) */
4 Module_cmstp BIT(01), /* SYSTEM(CMSTP) */
4 Module_mvs BIT(01), /* SYSTEM(MVS) */
4 Module_tso BIT(01), /* SYSTEM(TSO) */
4 Module_cics BIT(01), /* SYSTEM(CICS) */
4 Module_ims BIT(01), /* SYSTEM(IMS) */
4 Module_sysx BIT(02), /* Reserved */

2 Module_blocks FIXED BIN(31), /* Count of blocks */
/*
2 Module_size FIXED BIN(31), /* Size of module */
/*
2 Module_shooks FIXED BIN(31), /* Count of stmt hooks */
/*
2 Module_phooks FIXED BIN(31), /* Count of path hooks */
/*
2 Module_data_end CHAR(0);

DECLARE
1 Block_table BASED(A_block_table),
2 Block_a_data POINTER,
2 Block_count FIXED BIN(31),
2 Block_data(Blocks REFER(Block_count)),
3 Block_offset FIXED BIN(31),
3 Block_size FIXED BIN(31),
3 Block_level FIXED BIN(15),
3 Block_parent FIXED BIN(15),
3 Block_child FIXED BIN(15),
3 Block_sibling FIXED BIN(15),
3 Block_name CHAR(34) VARYING,
2 Block_table_end CHAR(0);

DECLARE
1 Stmt_table BASED(A_stmt_table),
2 Stmt_a_data POINTER,
2 Stmt_count FIXED BIN(31),
2 Stmt_data(Stmts REFER(Stmt_count)),
3 Stmt_offset FIXED BIN(31),
3 Stmt_no FIXED BIN(31),
3 Stmt_lineno FIXED BIN(15),
3 Stmt_reserved FIXED BIN(15),
3 Stmt_type FIXED BIN(15),
3 Stmt_block FIXED BIN(15),
2 Stmt_table_end CHAR(0);

```

---

Figure 76 (Part 3 of 7). Sample Facility 3: CEEBINT Module

---

```

DECLARE
  1 Path_table      BASED(A_path_table),
  2 Path_a_data     POINTER,
  2 Path_count      FIXED BIN(31),
  2 Path_data(Path  REFER(Path_count)),
  3 Path_offset     FIXED BIN(31),
  3 Path_no         FIXED BIN(31),
  3 Path_lineno     FIXED BIN(15),
  3 Path_reserved   FIXED BIN(15),
  3 Path_type       FIXED BIN(15),
  3 Path_block      FIXED BIN(15),
  2 Path_table_end  CHAR(0);

DECLARE Blocks      FIXED BIN(31);
DECLARE Stmts      FIXED BIN(31);
DECLARE Paths      FIXED BIN(31);
DECLARE Visits     FIXED BIN(31);

DECLARE A_block_table  POINTER;
DECLARE A_stmt_table   POINTER;
DECLARE A_path_table   POINTER;

DECLARE Addr         BUILTIN;

Declare IBMBHKS external entry( fixed bin(31), fixed bin(31));

Dcl HksFncStmt  fixed bin(31) init(1) static;
Dcl HksFncEntry fixed bin(31) init(2) static;
Dcl HksFncExit  fixed bin(31) init(3) static;
Dcl HksFncPath  fixed bin(31) init(4) static;
Dcl HksFncLabel fixed bin(31) init(5) static;
Dcl HksFncBCall fixed bin(31) init(6) static;
Dcl HksFncACall fixed bin(31) init(7) static;
Dcl HksRetCode  fixed bin(31);

DECLARE IBMSIR      EXTERNAL ENTRY ( POINTER );

/*****
/*
/* End of structures and variables for use in invoking IBMSIR  */
/*
*****/

```

---

Figure 76 (Part 4 of 7). Sample Facility 3: CEEBINT Module

---

```

/*****
/*
/* Sample code for invoking IBMBSIR
/*
/* IBMBSIR is first invoked to get the block and hook counts.
/*
/* If that invocation is successful, the block table is allocated
/* and IBMBSIR is invoked to fill in that table.
/*
/* If that invocation is also successful, the path hook table is
/* allocated and IBMBSIR is invoked to fill in that table.
/*
/*
/*****

Sir_fnccode = 3;

/*- If counts are to be obtained from a non-MAIN routine  -*/
/*- then put comments around the "Entryaddr(Sir_entry) =  -*/
/*- A_Main" statement and remove the comments from      -*/
/*- around the DCL and the "Sir_entry = P00" statements  -*/
/*- and change P00 (both places) to the name of the     -*/
/*- non-MAIN routine:                                   -*/

    Entryaddr(Sir_entry) = A_main;
/*- DCL P00 External entry;                               -*/
/*- Sir_entry = P00;                                     -*/

Sir_mod_data = Addr(Module_data);

Call IBMBSIR( Addr(Sir_data) );

If Sir_retcde = 0 then
Do;
    Sir_fnccode = 4;

    Blocks = Module_blocks;
    Allocate Block_table;
    Block_a_data = Addr(Block_data);
    Sir_a_data = A_block_table;

    Call IBMBSIR( Addr(Sir_data) );

    If Sir_retcde = 0 then
    Do;
        Sir_fnccode = 6;

        Paths = Module_phooks;
        Allocate Path_table;
        Path_a_data = Addr(Path_data);
        Sir_a_data = A_path_table;

        Call IBMBSIR( Addr(Sir_data) );
    End;
Else
Do ;
    Put skip list('CEEBINT MSG_2: Sir_retcde was not zero');
    Put skip list('CEEBINT MSG_2: Sir_retcde = ',Sir_retcde);
    Put skip list('CEEBINT MSG_2: Path table not allocated');
End;
End;

```

---

Figure 76 (Part 5 of 7). Sample Facility 3: CEEBINT Module

```

Else
  Do ;
    Put skip list('CEEBINT MSG_1: Sir_retcode was not zero');
    Put skip list('CEEBINT MSG_1: Sir_retcode = ',Sir_retcode);
    Put skip list('CEEBINT MSG_1: Block table not allocated!');
    Put skip list('CEEBINT MSG_1: Path table not allocated!');
  End;

/*****
/*
/* End of sample code for invoking IBMBSIR
/*
/*
*****/

/*****
/*
/* Sample code for setting up hook exit
/*
/* The first two sets of instructions merely create an exit list
/* containing one element and create the hook exit block to which
/* that element will point. Note that the hook exit is enabled
/* by this code, but it is not activated since Hook_exit_rtn = 0.
/*
/*
*****/

Allocate Exit_list;
A_Exits = A_exit_list;
Exit_list_count = 1;

Allocate Hook_exit_block;
Hook_exit_len = Stg(Hook_Exit_block);
Hook_exit_userword = UserWd;
Hook_exit_rtn = Sysnull();
Hook_exit_ptr = Sysnull();

/*****
/*
/*
/* The following code will cause the hook exit to invoke a
/* routine called HOOKUP that will keep count of how often each
/* path hook in the MAIN routine is executed.
/*
/*
/* First, the address of HOOKUP is put into the slot for the
/* address of the routine to be invoked when each hook is hit.
/*
/*
/* Then, pointers to the block table and the path hook table
/* obtained from IBMBSIR in the sample code above are put into
/* the exit area. Next, the address of the routine being
/* monitored, in this case the MAIN routine, is put into the
/* exit area. Additionally, a table to keep count of the number
/* of visits is created, and its address is also put into the
/* exit area.
/*
/*
/* Finally IBMHKS is invoked to turn on the PATH hooks.
/*
/*
*****/

If Sir_retcode = 0 then
  Do;
    Entry_var = HOOKUP;
    Hook_exit_rtn = Addr(Entry_var)->Based_ptr;

    Allocate Exit_area;

```

Figure 76 (Part 6 of 7). Sample Facility 3: CEEBINT Module

---

```

Exit_bdata = A_block_table;
Exit_pdata = A_path_table;

/*- If counts are to be obtained from a non-MAIN  -*/
/*- routine then replace "A_Main" in the following -*/
/*- statement with the name of the non-MAIN routine -*/
/*- (the name declared earlier as EXTERNAL ENTRY). -*/

Exit_epa = A_main;

Visits = Paths;
Allocate Hook_table;
Hook_visits = 0;

Call IBMBHKS( HksFncPath, HksRetCode );
End;
Else
Do ;
Put skip list('CEEBINT MSG_1: Sir_retcode was not zero');
Put skip list('CEEBINT MSG_1: Sir_retcode = ',Sir_retcode);
Put skip list('CEEBINT MSG_1: Exit area not allocated');
Put skip list('CEEBINT MSG_1: Hook table not allocated');
End;

/*****
/*
/* End of sample code for setting up hook exit
/*
*****/

/*****
/*
/* Place your actual user code here
/*
*****/
Put skip list('CEEBINT: At end of CEEBINT initialization');

END CEEBINT;

```

---

Figure 76 (Part 7 of 7). Sample Facility 3: CEEBINT Module

---

```

%PROCESS TEST(NONE,NOSYM) LANGLVL(SPROG);
Hookup : PROC( exit_for_hooks );

DECLARE Exit_for_hooks Pointer;

DECLARE
  1 Hook_exit_block  BASED(Exit_for_hooks),
  2 Hook_exit_len    FIXED BIN(31),
  2 Hook_exit_rtn    POINTER,
  2 Hook_exit_fnccode FIXED BIN(31),
  2 Hook_exit_retcode FIXED BIN(31),
  2 Hook_exit_rsncode FIXED BIN(31),
  2 Hook_exit_userword POINTER,
  2 Hook_exit_ptr    POINTER,
  2 Hook_exit_reserved POINTER,
  2 Hook_exit_dsa    POINTER,
  2 Hook_exit_addr   POINTER,
  2 Hook_exit_end    CHAR(0);

DECLARE
  1 Exit_area        BASED(Hook_exit_ptr),
  2 Exit_bdata       POINTER,
  2 Exit_pdata       POINTER,
  2 Exit_epa         POINTER,
  2 Exit_a_visits    POINTER,
  2 Exit_area_end    CHAR(0);

DECLARE
  1 Block_table      BASED(exit_bdata),
  2 Block_a_data     POINTER,
  2 Block_count      FIXED BIN(31),
  2 Block_data(Blocks REFER(Block_count)),
  3 Block_offset     FIXED BIN(31),
  3 Block_size       FIXED BIN(31),
  3 Block_level      FIXED BIN(15),
  3 Block_parent     FIXED BIN(15),
  3 Block_child      FIXED BIN(15),
  3 Block_sibling    FIXED BIN(15),
  3 Block_name       CHAR(34) VARYING,
  2 Block_table_end  CHAR(0);

DECLARE
  1 Path_table       BASED(exit_pdata),
  2 Path_a_data      POINTER,
  2 Path_count       FIXED BIN(31),
  2 Path_data(Paths REFER(Path_count)),
  3 Path_offset      FIXED BIN(31),
  3 Path_no          FIXED BIN(31),
  3 Path_lineno      FIXED BIN(15),
  3 Path_reserved    FIXED BIN(15),
  3 Path_type        FIXED BIN(15),
  3 Path_block       FIXED BIN(15),
  2 Path_table_end   CHAR(0);

DECLARE
  1 Hook_table       BASED(Exit_a_visits),
  2 Hook_data_size   FIXED BIN(31),
  2 Hook_data(Visits REFER(Hook_data_size)),
  3 Hook_visits      FIXED BIN(31),
  2 Hook_data_end    CHAR(0);

DECLARE Ps          FIXED BIN(31);
DECLARE Ix          FIXED BIN(31);
DECLARE Jx          FIXED BIN(31);

```

---

Figure 77 (Part 1 of 5). Sample Facility 3: HOOKUP Program

---

```

/* ----- Notes on cpu timing ----- */
/*                                     */
/* TIMCPU is an assembler routine (TIMCPUVM is the CMS */
/* filename) which uses DIAG to get the cpu time in */
/* microseconds since the last system reset. Since */
/* only the low order 31-bits is returned to this */
/* procedure it is possible that a negative number may */
/* result when subtracting the after_time from the */
/* before time.                                     */

/* ----- Declares for cpu timing ----- */
DCL TIMINI          ENTRY OPTIONS (ASSEMBLER INTER);
DCL TIMCPU          ENTRY OPTIONS (ASSEMBLER INTER);
DCL Decimal         BUILTIN;
DCL cpu_visits      FIXED BIN(31) STATIC; /* = visits */
DCL Hook_cpu_visits Pointer STATIC ;
DCL exe_mon         FIXED BIN(31) EXTERNAL;
/* exe_mon <=0 no cpu timing data is printed */
/* exe_mon =1 only block data is obtained - default */
/* exe_mon >=2 all hooks are timed */
DECLARE
  1 Hook_time_table    BASED(Hook_cpu_visits),
  2 Hooksw,
    3 BLK_entry_time    FIXED BIN(31,0),
    3 Before            FIXED BIN(31,0),
    3 After             FIXED BIN(31,0),
    3 Temp_cpu          FIXED BIN(31,0),
  2 Hook_time_data_size FIXED BIN(31),
  2 Hook_time_data(cpu_visits Refer(Hook_time_data_size)),
    3 Hook_time         Fixed Bin(31),
  2 Hook_time_data_end Char(0);
/* ----- END Declares for cpu timing ----- */

/* compute offset */
ps = hook_exit_addr - exit_epa;

/* search for hook */
do ix = 1 to path_count
  while ( ps -= path_offset(ix) );
end;

/* if hook found - update table */
if (ix > 0) & (ix <=path_count) then do;
  hook_visits(ix) = hook_visits(ix) + 1;

/* ----- this SELECT gets the cpu timings ----- */
Select;
  When(path_type(ix)=1 & ix=1) do; /* external block entry */
    exe_mon = 1; /* default can be overridden in Main */
    cpu_visits = hook_data_size; /* number of hooks */
    allocate hook_time_table; /* cpu time table */
    hook_time = 0; /* initialize the table */
    call TIMINI; /* initialize timers */
    call TIMCPU (BLK_entry_time); /* get block entry time */
    hooksw.before = BLK_entry_time; /* block entry */
    hooksw.after = BLK_entry_time; /* block entry */
  end; /* end of the When(1) do */

```

---

Figure 77 (Part 2 of 5). Sample Facility 3: HOOKUP Program

---

```

Otherwise do;
  if (exe_mon > 0) then do;
    /* collect cpu time data */
    /* get CPU time in microseconds since last call */
    call TIMCPU (hooksw.after);
    temp_cpu = hooksw.after - hooksw.before;
    hook_time(ix) = hook_time(ix) + temp_cpu;
    hooksw.before = hooksw.after;
  end; /* end of the if exe_mon > 0 then do */
end; /* end of the otherwise do */
End; /* end of the Select */
/* ---End of the SELECT to get the cpu timings ----- */
end; /* end of the if (ix > 0) & (ix <=path_count) then do */
else;

/* if exit from external (main), report */
if path_type(ix) = 2 & path_block(ix) = 1 then
  Do;
    call post_hook_processing;
    free hook_time_table;
  End;
else;

/* don't invoke the debugging tool */
hook_exit_retcode = 4;
hook_exit_rsncode = 0;
/* ----- */
Post_hook_processing: Procedure;
/* ----- */
DECLARE Total_v_count      FLOAT DEC(06);
DECLARE Total_c_count      FLOAT DEC(06);
DECLARE Tot_vst_per_blk    FIXED BIN(31,0);
DECLARE Tot_v_pcnt_per_blk FIXED DEC(6,4);
DECLARE Tot_cpu_per_blk    FIXED BIN(31,0);
DECLARE Tot_c_pcnt_per_blk FIXED DEC(6,4);
DECLARE Percent_v          FIXED DEC(6,4);
DECLARE Percent_c          FIXED DEC(6,4);
DECLARE Col1               CHAR(30);
DECLARE Col2               CHAR(14);
DECLARE Col2_3             CHAR(30);
DECLARE Col3               CHAR(10);
DECLARE Col4               CHAR(14);
DECLARE Col5               CHAR(14);
DECLARE Col4_5             CHAR(28);
DECLARE Millisec_out       FIXED DEC(10,3);

total_v_count = 0;
total_c_count = 0;

do jx = 1 to path_count; /* get total hook_visits */
  total_v_count = total_v_count + hook_visits(jx);
  total_c_count = total_c_count + hook_time (jx);
end;

do Jx = 1 to path_count;
  if path_type(jx) = 1 then
  do; /* at block entry so print (block) headers */
    put skip list( ' ');
    put skip list( 'Data for block ' ||
      block_name(path_block(jx)) );
    put skip list( ' ');
    col1 = ' Statement Type';
    col2_3 = ' --- Visits ---';
    if exe_mon > 0
      then col4_5 = '--- CPU Time ---';
      else col4_5 = ' ';
    put skip list( col1 || ' ' || col2_3 || ' ' || col4_5 );
  end;
end;

```

---

Figure 77 (Part 3 of 5). Sample Facility 3: HOOKUP Program

```

col1 = ' ';
col2 = '      Number';
col3 = ' Percent';
if exe_mon > 0 then
do;
  col4 = ' Milliseconds';
  col5 = ' Percent';
end;
else
do;
  col4 = ' ';
  col5 = ' ';
end;
put skip list( col1 || ' ' || col2 || ' ' || col3 ||
              ' ' || col4 || col5);
put skip list( ' ' );
end;
else;

Select ( path_type(jx) );
  when ( 1 )
    col1 = Substr(char(path_no(jx)),4,11) || ' block entry';
  when ( 2 )
    col1 = Substr(char(path_no(jx)),4,11) || ' block exit';
  when ( 3 )
    col1 = Substr(char(path_no(jx)),4,11) || ' label';
  when ( 4 )
    col1 = Substr(char(path_no(jx)),4,11) || ' before call';
  when ( 5 )
    col1 = Substr(char(path_no(jx)),4,11) || ' after call';
  when ( 6 )
    col1 = Substr(char(path_no(jx)),4,11) || ' start of do loop';
  when ( 7 )
    col1 = Substr(char(path_no(jx)),4,11) || ' if true';
  when ( 8 )
    col1 = Substr(char(path_no(jx)),4,11) || ' if false';
  otherwise
    col1 = Substr(char(path_no(jx)),4,11);
end; /* end of the select */

col2 = hook_visits(jx);
percent_v = 100* (hook_visits(jx)/total_v_count) ;
percent_c = 100* (hook_time (jx)/total_c_count) ;

/* ----- print out the cpu timings ----- */
if exe_mon <= 0 then /* no cpu time wanted */
  put skip list ( col1 || ' ' || col2 || ' ' || percent_v);
else do; /* compute and print cpu times */
  Select;
  When (path_type(jx) = 1 & jx = 1) Do;
    /* at external block entry */
    /* initialize block counters */
    tot_vst_per_blk = hook_visits(jx);
    tot_v_pcnt_per_blk = percent_v;
    tot_cpu_per_blk = 0;
    tot_c_pcnt_per_blk = 0;
    /* no CPU time on ext block entry line */
    put skip list ( col1 || ' ' || col2 || ' ' || percent_v);
  End; /* End of the When (... = 1 & jx = 1) */

```

Figure 77 (Part 4 of 5). Sample Facility 3: HOOKUP Program

---

```

When (path_type(jx) = 2) Do;
/* @ block exit so print cpu summary line */
/* write the "block exit" line */
tot_vst_per_blk = tot_vst_per_blk + hook_visits(jx);
tot_v_pcnt_per_blk = tot_v_pcnt_per_blk + percent_v;
tot_cpu_per_blk = tot_cpu_per_blk + hook_time (jx);
tot_c_pcnt_per_blk = tot_c_pcnt_per_blk + percent_c;
if exe_mon > 1 then do;
    millisec_out = Decimal(hook_time(jx),11,3)/1000;
    put skip list ( col1 || ' ' || col2 || ' ' || percent_v
        || ' ' || millisec_out || ' ' || percent_c);
end;
Else do; /* only want cpu time on summary line */
    put skip list ( col1 || ' ' || col2 || ' ' || percent_v);
end;
/* write the "Totals for Block" line */
col1 = '      Totals for block';
col2 = tot_vst_per_blk;
millisec_out = Decimal(tot_cpu_per_blk,11,3) / 1000;
put skip list ( col1 || ' ' || col2 || ' ' ||
    tot_v_pcnt_per_blk || ' ' || millisec_out || ' ' ||
    tot_c_pcnt_per_blk);
End; /* End of the When (path_type(jx) = 2) */

Otherwise do; /* cpu hook timing */
if path_type(jx) = 1 then do;
    /* at internal block entry */
    /* initialize block counters */
    tot_vst_per_blk = hook_visits(jx);
    tot_v_pcnt_per_blk = percent_v;
    tot_cpu_per_blk = hook_time(jx);
    tot_c_pcnt_per_blk = percent_c;
end;
else do;
    /* not at block entry */
    tot_vst_per_blk = tot_vst_per_blk + hook_visits(jx);
    tot_v_pcnt_per_blk = tot_v_pcnt_per_blk + percent_v;
    tot_cpu_per_blk = tot_cpu_per_blk + hook_time (jx);
    tot_c_pcnt_per_blk = tot_c_pcnt_per_blk + percent_c;
end;
if exe_mon > 1 then do;
    millisec_out = Decimal(hook_time(jx),11,3)/1000;
    put skip list ( col1 || ' ' || col2 || ' ' || percent_v
        || ' ' || millisec_out || ' ' || percent_c);
end; /* end of the if exe_mon > 1 then do */
else /* only want total cpu time for the block */
    put skip list ( col1 || ' ' || col2 || ' ' || percent_v);
End; /* End of the Otherwise Do */
End; /* End of the Select */
end; /* end of the else do */
end; /* end of the do Jx = 1 to path_count */

put skip list( ' ');

END post_hook_processing;
END Hookup;

```

---

Figure 77 (Part 5 of 5). Sample Facility 3: HOOKUP Program

---

```

TIMINI  CSECT
* MVS Version
* This program will initialize the interval timer for
* measuring the task execution (CPU) time so that calls to
* the TIMCPU routine will return the proper result. Reg 0
* will contain the value of 0 at exit of this program.
* No arguments are passed to this program.
*
* STANDARD PROLOGUE
  STM  14,12,12(13)      Save caller's registers
  LR   12,15             Get base address
  USING TIMINI,12       Establish addressability
  ST   13,SAVE+4        Forward link of save areas
  LA   10,SAVE          Our save area address
  ST   10,8(,13)        Backward link of save areas
  LR   13,10            Our save area is now the current one
  STIMER TASK,MICVL=TIME Set interval timer (MVS)
*
* STANDARD EPILOG
EXIT   L   13,4(,13)     Get previous save area address
      L   14,12(,13)    Restore register 14
      LM  1,12,24(13)   Restore regs 1 - 12
      SR  0,0           Return value = 0
      MVI 12(13),X'FF'  Flag return
      SR  15,15        Return code = 0
      BR  14           Return to caller
      DS  0D
TIME  DC  XL8'000007FFFFFFFF000' Initial interval timer
      *               time (MVS)
SAVE  DS  18F'0'       Our save area
      END  TIMINI

```

---

Figure 78. Sample Facility 3: TIMINI Module for MVS

---

```

TIMINI  CSECT
*
      SR  0,0           Return value = 0
      SR  15,15        Return code = 0
      BR  14           Return to caller
*
      END  TIMINI

```

---

Figure 79. Sample Facility 3: TIMINI Module for VM

---

```

TIMCPU  CSECT
*  MVS version
*  This program will get the amount of time, in microseconds,
*  remaining in the interval timer that has been previously
*  set by the TIMINI routine. The result is placed in the
*  full-word argument that is passed to this program.
*
*  STANDARD PROLOGUE
      STM 14,12,12(13)    Save caller's registers
      LR 12,15            Get base address
      USING TIMCPU,12    Establish addressability
      ST 13,SAVE+4       Forward link of save areas
      LA 10,SAVE         Our save area address
      ST 10,8(,13)       Backward link of save areas
      LR 13,10           Our save area is now the current one
*
      L 5,0(,1)          Get address of parameter
      TTIMER ,MIC,CPU    Get the remaining time and save it
*
      L 2,CPU            Get high order part ...
      L 3,CPU+4          and low order part ...
      SLDL 2,20          and only keep time (no date)
      L 0,INITTIME       Get initial timer value
      SR 0,2             Subtract to get elapsed CPU time
      ST 0,0(,5)        Put CPU time into parameter
*
*  STANDARD EPILOG
EXIT  L 13,4(,13)       Get previous save area address
      L 14,12(,13)      Restore register 14
      LM 1,12,24(13)    Restore registers 1 - 12
*                          (r0 has time)
      MVI 12(13),X'FF'   Flag return
      SR 15,15          Return code = 0
      BR 14             Return to caller
      DS 0D
CPU   DS CL8           Area to store remaining time
INITTIME DC XL4'7FFFFFFF' Initial timer value
SAVE  DS 18F'0'        Our save area
      END  TIMCPU

```

---

Figure 80. Sample Facility 3: TIMCPU Module for MVS

---

```

TIMCPU  CSECT
*  CMS version
*  This program will get the amount of time, in microseconds,
*  remaining in the interval timer that has been previously
*  set by the TIMINI routine and will return the result to
*  the calling routine. The calling routine calls this
*  routine as a procedure, passing one fullword argument.
*
*  STANDARD PROLOGUE
      STM 14,12,12(13)    Save caller's registers
      LR 12,15           Get base address
      USING TIMCPU,12    Establish addressability
      ST 13,SAVE+4       Forward link save areas
      LA 10,SAVE         Our save area address
      ST 10,8(,13)      Backward link save areas
      LR 13,10           Our save area is now the current one
*
      L   5,0(,1)        Get address of parameter
*
      LA 1,DATETIME      Address of CMS pseudo timer info
      DIAG 1,0,X'C'      Get the time from CMS
      L   0,CPU+4         and low order part
      ST 0,0(,5)         Put CPU time into parameter
*
*  STANDARD EPILOG
EXIT  L   13,4(,13)      Get previous save area address
      L   14,12(,13)     Restore register 14
      LM 1,12,24(13)     Restore registers 1 - 12
*                          (r0 has time)
      SR 15,15           Return code = 0
      BR 14              Return to caller
      DS 00
DATETIME DS 0CL32       CMS date and time info
DATE     DS CL8         Date mm/dd/yy
TOD      DS CL8         Time of day HH:MM:SS
CPU      DS CL8         Virtual CPU time
TOTCPU   DS CL8         Total CPU time
SAVE     DS 18F'0'      Our save area
END      TIMCPU

```

---

Figure 81. Sample Facility 3: TIMCPU Module for VM

---

## Chapter 14. Efficient Programming

This chapter describes methods for improving the efficiency of PL/I programs.

The section titled “Efficient Performance” suggests how to tune run-time performance of PL/I programs.

The “Global Optimization Features” section discusses the various types of global optimization performed by the compiler when OPTIMIZE(TIME) is specified. The section also contains some hints on coding PL/I programs to take advantage of global optimization.

Finally, pages 323 through 345 cover performance-related topics. Each of these sections has a title that begins “Notes about... .” They cover the following topics:

- Data elements
- Expressions and references
- Data conversion
- Program organization
- Recognition of names
- Storage control
- General statements
- Subroutines and functions
- Built-in functions and pseudovariables
- Input and output
- Record-oriented data transmission
- Stream-oriented data transmission
- Picture specification characters
- Condition handling.
- Multitasking.

---

### Efficient Performance

Because of the modularity of the PL/I libraries and the wide range of optimization performed by the compiler, many PL/I application programs have acceptable performance and do not require tuning.

This section suggests ways in which you can improve the performance of programs that do not fall into the above category. Other ways to improve performance are described later in this chapter under the notes for the various topics.

It is assumed that you have resolved system considerations (for example, the organization of the PL/I libraries), and also that you have some knowledge of the compile-time and run-time options (see Chapter 1, “Using Compile-Time Options and Facilities” and *Language Environment for MVS & VM Programming Guide*).

### Tuning a PL/I Program

Remove all debugging aids from the program.

The overhead incurred by some debugging aids is immediately obvious because these aids produce large amounts of output. However, debugging aids such as the SUBSCRIPTRANGE and STRINGRANGE condition prefixes which produce output

only when an error occurs, also significantly increase both the storage requirements and the execution time of the program.

You should also remove PUT DATA statements from the program, especially those for which no data list is specified. These statements require control blocks to describe the variables and library modules to convert the control block values to external format. Both of these operations increase the program's storage requirements.

Using the GOSTMT or GONUMBER compile-time option does not increase the execution time of the program, but will increase its storage requirements. The overhead is approximately 4 bytes per PL/I statement for GOSTMT, and approximately 6 bytes per PL/I statement for GONUMBER.

Specify run-time options in the PLIXOPT variable, rather than as parameters passed to the program initialization routines. It might prove beneficial to alter existing programs to take advantage of the PLIXOPT variable, and to recompile them. For a description of using PLIXOPT, see the *Language Environment for MVS & VM Programming Guide*.

After removing the debugging aids, compile and run the program with the RPTSTG run-time option. The output from the RPTSTG option gives the size that you should specify in the STACK run-time option to enable all PL/I storage to be obtained from a single allocation of system storage. For a full description of the output from the RPTSTG option, see the *Language Environment for MVS & VM Programming Guide*.

**Manipulating Source Code:** Many operations are handled in-line. You should determine which operations are performed in-line and which require a library call, and to arrange your program to use the former wherever possible. The majority of these in-line operations are concerned with data conversion and string handling. (For conditions under which string operations are handled in-line, see Table 41 on page 328. For implicit data conversion performed in-line, see Table 42 on page 330. For conditions under which string built-in functions are handled in-line, see "In-Line Code" on page 314.)

In PL/I there are often several different ways of producing a given effect. One of these ways is usually more efficient than another, depending largely on the method of implementation of the language features concerned. The difference might be only one or two machine instructions, or it might be several hundred.

You can also tune your program for efficient performance by looking for alternative source code that is more efficiently implemented by the compiler. This will be beneficial in heavily used parts of the program.

It is important to realize, however, that a particular use of the language is not necessarily bad just because the correct implementation is less efficient than that for some other usage; it must be reviewed in the context of what the program is doing now and what it will be required to do in the future.

## Tuning a Program for a Virtual Storage System

The output of the compiler is well suited to the requirements of a virtual storage system. The executable code is read-only and is separate from the data, which is itself held in discrete segments. For these reasons there is usually little cause to tune the program to reduce paging. Where such action is essential, there are a number of steps that you can take. However, keep in mind that the effects of tuning are usually small.

The object of tuning for a virtual storage system is to minimize the paging; that is, to reduce the number of times the data moves from auxiliary storage into main storage and vice-versa. You can do this by making sure that items accessed together are held together, and by making as many pages as possible read-only.

When using the compiler, you can write the source program so that the compiler produces the most advantageous use of virtual storage. If further tuning is required, you can use linkage editor statements to manipulate the output of the compiler so that certain items appear on certain pages.

By designing and programming modular programs, you can often achieve further tuning of programs for a virtual storage system.

To enable the compiler to produce output that uses virtual storage effectively, take care both in writing the source code and in declaring the data. When you write source code, avoid large branches around the program. Place statements frequently executed together in the same section of the source program.

When you declare data, your most important consideration should be the handling of data aggregates that are considerably larger than the page size. You should take care that items within the aggregate that are accessed together are held together. In this situation, the choice between arrays of structures and structures of arrays can be critical.

Consider an aggregate containing 3000 members and each member consisting of a name and a number. If it is declared:

```
DCL 1 A(3000),  
  2 NAME CHAR(14),  
  2 NUMBER FIXED BINARY;
```

the 100th name would be held adjacently with the 100th number and so they could easily be accessed together.

However, if it is declared:

```
DCL 1 A,  
  2 NAME(3000) CHAR(14),  
  2 NUMBER(3000) FIXED BINARY;
```

all the names would be held contiguously followed by all the numbers, thus the 100th name and the 100th number would be widely separated.

When choosing the storage class for variables, there is little difference in access time between `STATIC INTERNAL` or `AUTOMATIC`. The storage where both types of variable are held is required during execution for reasons other than access to the variables. If the program is to be used by several independent users simultaneously, declare the procedure `REENTRANT` and use `AUTOMATIC` to provide separate copies of the variables for each user. The storage used for based

or controlled variables is not, however, required and avoiding these storage classes can reduce paging.

You can control the positioning of variables by declaring them `BASED` within an `AREA` variable. All variables held within the area will be held together.

A further refinement is possible that increases the number of read-only pages. You can declare `STATIC INITIAL` only those variables that remain unaltered throughout the program and declare the procedure in which they are contained `REENTRANT`. If you do this, the static internal CSECT produced by the compiler will be made read-only, with a consequent reduction in paging overhead.

The compiler output is a series of CSECTs (control sections). You can control the linkage editor so that the CSECTs you specify are placed together within pages, or so that a particular CSECT will be placed at the start of a page. The linkage editor statements you need to do this are given in your Linkage Editor and Loader publication.

The compiler produces at least two CSECTs for every external procedure. One CSECT contains the executable code and is known as the *program CSECT*; the other CSECT contains addressing data and static internal variables and is known as the *static CSECT*. In addition, the compiler produces a CSECT for every static external variable. A number of other CSECTs are produced for storage management. A description of compiler output is given in *Language Environment for MVS & VM Debugging Guide and Run-Time Messages*.

You can declare variables `STATIC EXTERNAL` and make procedures external, thus getting a CSECT for each external variable and a program CSECT and a static internal CSECT for each external procedure. It is possible to place a number of variables in one CSECT by declaring them `BASED` in an `AREA` that has been declared `STATIC EXTERNAL`.

When you have divided your program into a satisfactory arrangement of CSECTs, you can then analyze the use of the CSECTs and arrange them to minimize paging. You should realize, however, that this can be a difficult and time-consuming operation.

---

## Global Optimization Features

The PL/I compiler attempts to generate object programs that run rapidly. In many cases, the compiler generates efficient code for statements in the sequence written by the programmer. In other cases, however, the compiler might alter the sequence of statements or operations to improve the performance, while producing the same result.

The compiler carries out the following types of optimization:

- Expressions:
  - Common expression elimination
  - Redundant expression elimination
  - Simplification of expressions.
- Loops:
  - Transfer of expressions from loops
  - Special-case code for `DO` statements.

- Arrays and structures:
  - Initialization
  - Assignments
  - Elimination of common control data.
- In-line code for:
  - Conversions
  - RECORD I/O
  - String manipulation
  - Built-in functions.
- Input/output:
  - Key handling for REGIONAL data sets
  - Matching format lists with data lists.
- Other:
  - Library subroutines
  - Use of registers
  - Analyzing run-time options during compile time (the PLIXOPT variable).

PL/I performs some of these types of optimization even when the NOOPTIMIZE option is specified. PL/I attempts full optimization, however, only when a programmer specifies the OPTIMIZE (TIME) compile-time option.

## Expressions

The following sections describe optimization of expressions.

### Common Expression Elimination

A *common expression* is an expression that occurs more than once in a program, but is intended to result in the same value each time it is evaluated. A *common expression* is also an expression that is identical to another expression, with no intervening modification to any operand in the expression. The compiler eliminates a common expression by saving the value of the first occurrence of the expression either in a temporary (compiler generated) variable, or in the program variable to which the result of the expression is assigned. For example:

```
X1 = A1 * B1;
:
Y1 = A1 * B1;
```

Provided that the values of A1, B1, and X1 do not change between the processing of these statements, the statements can be optimized to the equivalent of the following PL/I statements:

```
X1 = A1 * B1;
:
Y1 = X1;
```

Sometimes the first occurrence of a common expression involves the assignment of a value to a variable that is modified before it occurs in a later expression. In this case, the compiler assigns the value to a temporary variable. The example given above becomes:

```
Temp = A1 * B1;  
X1 = Temp;  
:  
X1 = X1 + 2;  
:  
Y1 = Temp;
```

If the common expression occurs as a subexpression within a larger expression, the compiler creates a temporary variable to hold the value of the common subexpression. For example, in the expression  $C1 + A1 * B1$ , a temporary variable contains the value of  $A1 * B1$ , if this were a common subexpression.

An important application of this technique occurs in statements containing subscripted variables that use the same subscript value for each variable. For example:

```
PAYTAX(MNO)=PAYCODE(MNO)*WKPMNT(MNO);
```

The compiler computes the value of the subscript MNO only once, when the statement processes. The computation involves the conversion of a value from decimal to binary, if MNO is declared to be a decimal variable.

### Redundant Expression Elimination

A *redundant expression* is an expression that need not be evaluated for a program to run correctly. For example, the logical expression:

```
(A=D) | (C=D)
```

contains the subexpressions (A=D) and (C=D). The second expression need not be evaluated if the first one is true. This optimization makes using a compound logical expression in a single IF statement more efficient than using an equivalent series of nested IF statements.

### Simplification of Expressions

There are two types of expression simplification processes explained below.

**Modification of Loop Control Variables:** Where possible, the expression-simplification process modifies both the control variable and the iteration specification of a do-loop for more efficient processing when using the control variable as a subscript. The compiler calculates addresses of array elements faster by replacing multiplication operations with addition operations. For example, the loop:

```
Do I = 1 To N By 1;  
  A(I) = B(I);  
End;
```

assigns N element values from array B to corresponding elements in array A. Assuming that each element is 4 bytes long, the address calculations used for each iteration of the loop are:

Base(A)+(4\*I) for array A, and  
Base(B)+(4\*I) for array B,

where Base represents the base address of the array in storage. The compiler can convert repeated multiplication of the control variable by a constant (that represents the length of an element) to faster addition operations. It converts the optimized DO statement above into object code equivalent to the following statement:

```
Do Temp = 4 By 4 To 4*N;
```

The compiler converts the element address calculations to the equivalent of:

Base(A) + Temp for array A, and  
Base(B) + Temp for array B.

This optimization of a loop control variable and its iteration can occur only when the control variable (used as a subscript) increases by a constant value. Programs should not use the value of the control variable outside the loop in which it controls iteration.

**Defactorization:** Where possible, a constant in an array subscript expression is an offset in the address calculation. For example, PL/I calculates the address of a 4-byte element:

A(I+10)

as:

(Base(A)+4\*10)+I\*4

### Replacement of Constant Expressions

Expression simplification replaces constant expressions of the form A+B or A\*B, where A and B are integer constants, with the equivalent constant. For example, the compiler replaces the expression 2+5 with 7.

**Replacement of Constant Multipliers and Exponents:** The expression-simplification process replaces certain constant multipliers and exponents. For example:

A\*2 becomes A+A,

and

A\*\*2 becomes A\*A.

**Elimination of Common Constants:** If a constant occurs more than once in a program, the compiler stores only a single copy of that constant. For example, in the following statements:

```
Week_No = Week_No + 1;  
Record_Count = Record_Count + 1;
```

the compiler stores the 1 only once, provided that Week\_No and Record\_Count have the same attributes.

**Code for Program Branches:** The compiler allocates base registers for branch instructions in the object program in accordance with the logical structure of the program. This minimizes the occurrence of program-addressing load instructions in the middle of deeply nested loops.

Also, the compiler arranges the branch instructions generated for IF statements as efficiently as possible. For example, a statement such as:

```
IF condition THEN GOTO label;
```

is defined by the PL/I language as being a test of *condition* followed by a branch on *false* to the statement following the THEN clause. However, when the THEN clause consists only of a GOTO statement, the statement compiles as a branch on *true* to the label specified in the THEN clause.

## Loops

In addition to the optimization described in “Modification of Loop Control Variables” on page 310, PL/I provides optimization features for loops as described in the following sections.

### Transfer of Expressions from Loops

Where it is possible to produce an error-free run without affecting program results, optimization moves invariant expressions or statements from inside a loop to a point that immediately precedes the loop. An expression or statement occurring within a loop is said to be *invariant* if the compiler can detect that the value of the expression or the action of the statement would be identical for each iteration of the loop. A loop can be either a do-loop or a loop in a program detectable by analyzing the flow of control within the program. For example:

```
Do I = 1 To N;  
  B(I) = C(I) * SQRT(N);  
  P = N * J;  
End;
```

This loop can be optimized to produce object code corresponding to the following statements:

```
Temp = SQRT(N);  
P = N * J;  
DO I = 1 TO N;  
  B(I) = C(I) * Temp;  
End;
```

If programmers want to use this type of optimization, they must specify REORDER for a BEGIN or PROCEDURE block that contains the loop. If a programmer does not specify the option, the compiler assumes the default option, ORDER, which inhibits optimization.

**Programming Considerations:** The compiler transfers expressions from inside to outside a loop on the assumption that every expression in the loop runs more frequently than expressions immediately outside the loop. Occasionally this assumption fails, and the compiler moves expressions out of loops to positions in which they run more frequently than if they had remained inside the loop. For example:

```
Do I = J To K While(X(I)=0);  
  X(I) = Y(I) * SQRT(N);  
End;
```

The compiler might move the expression SQRT(N) out of the loop to a position in which it is *possible* for the expression to be processed more frequently than in its

original position inside the loop. This undesired effect of optimization is prevented by using the ORDER option for the block in which the loop occurs.

The compiler detects loops by analyzing the flow of control. The compiler can fail to recognize a loop if programmers use label variables because of flowpaths that they know are seldom or never used. Using label variables can inadvertently inhibit optimization by making a loop undetectable.

### **Special Case Code for DO Statements**

Where possible for a do-loop, the compiler generates code in which the value of the control variable, and the values of the iteration specification, are contained in registers throughout loop execution. For example, the compiler attempts to maintain in registers the values of the variables I, K, and L in the following statement:

```
Do I = A To K By L;
```

This optimization uses the most efficient loop control instructions.

## **Arrays and Structures**

PL/I provides optimization for array and structure variables as described in the following sections.

### **Initialization of Arrays and Structures**

When arrays and some structures that have the BASED, AUTOMATIC, or CONTROLLED storage class are to be initialized by a constant specified in the INITIAL attribute, the compiler initializes the first element of the variable by the constant. The remainder of the initialization is a single move that propagates the value through all the elements of the variable. For example, for the following declaration:

```
DCL A(20,20) Fixed Binary Init((400)0);
```

the compiler initializes array A using this method.

### **Structure and Array Assignments**

The compiler implements structure and array assignment statements by single move instructions whenever possible. Otherwise, the compiler assigns values by the simplest loop possible for the operands in the declaration. For example:

```
DCL A(10),B(10), 1 S(10), 2 T, 2 U;  
  A=B;  
  A=T;
```

The compiler implements the first assignment statement by a single move instruction, and the second by a loop. This occurs because array T is interleaved with array U, thereby making a single move impossible.

### **Elimination of Common Control Data**

The compiler generates control information to describe certain program elements such as arrays. If there are two or more similar arrays, the compiler generates this descriptive information only once.

## In-Line Code

To increase efficiency, the PL/I compiler produces *in-line* code (code that it incorporates within programs) as a substitute for calls to generalized subroutines.

### In-line Code for Conversions

The compiler performs most conversions by in-line code, rather than by calls to the Library. The exceptions are:

- Conversions between character and arithmetic data
- Conversions from numeric character (PICTURE) data, where the picture includes characters other than 9, V, Z, or a single sign or currency character
- Conversions to numeric character (PICTURE) data, where the picture includes scale factors or floating point picture characters.

For example, the compiler converts data to PICTURE 'ZZ9V99' with in-line code.

### In-line Code for Record I/O

For consecutive buffered files under certain conditions, in-line code implements the input and output transmission statements READ, WRITE, and LOCATE rather than calls to the Library.

### In-line Code for String Manipulation

In-line code performs operations on many character strings (such as concatenation and assignment of adjustable, varying-length, and fixed-length strings). In-line code performs similar operations on many aligned bit strings that have a length that is a multiple of 8.

### In-line Code for Built-In Functions

The compiler uses in-line code to implement many built-in functions. INDEX and SUBSTR are examples of functions for which the compiler usually generates in-line code. TRANSLATE, VERIFY, and REPEAT are examples where the compiler generates in-line code for simple cases.

## Key handling for REGIONAL data sets

In certain circumstances, avoiding unnecessary conversions between fixed-binary and character-string data types simplifies key handling for REGIONAL data sets, as follows:

### REGIONAL(1)

If the key is a fixed binary integer with precision (12,0) through (23,0), there is no conversion from fixed binary to character string and back again.

### REGIONAL(2) and REGIONAL(3)

If the key is in the form K|I, where K is a character string and I is fixed binary with precision (12,0) through (23,0), the rightmost eight (8) characters of the resultant string do not reconvert to fixed binary. (This conversion would otherwise be necessary to obtain the region number.)

## Matching Format Lists with Data Lists

Where possible, the compiler matches format and data lists of edit-directed input/output statements at compile time. This is possible only when neither list contains repetition factors at compile time that are expressions with unknown values. This allows in-line code to convert to or from the data list item. Also, on input, PL/I can take the item directly from the buffer or, on output, place it directly into the buffer. This eliminates Library calls, except when necessary to transmit a block of data between the input or output device and the buffer. For example:

```
DCL (A,B,X,Y,Z) CHAR(25);  
Get File(SYSIN) Edit(X,Y,Z) (A(25));  
Put File(SYSPRINT) Edit(A,B) (A(25));
```

In this example, format list matching occurs at compile time; at run time, Library calls are required only when PL/I transmits the buffer contents to or from the input or output device.

## Run-time Library Routines

Language Environment for MVS & VM and PL/I library routines are packaged as a set, in such a manner that a link-edited object program contains only stubs that correspond to these routines. This packaging minimizes program main storage requirements. It can also reduce the time required for loading the program into main storage.

## Use of Registers

The compiler achieves more efficient loop processing by maintaining in registers the values of loop variables that are subject to frequent modification. Keeping values of variables in registers, as the flow of program control allows, results in considerable efficiency. This efficiency is a result of dispensing with time-consuming load-and-store operations that reset the values of variables in their storage locations. When, after loop processing, the latest value of a variable is not required, the compiler does not assign the value to the storage location of the variable as control passes out of the loop.

Specifying REORDER for the block significantly optimizes the allocation of registers. However, because the latest value of a variable can exist in a register but not in the storage location of that variable, the values of variables reset in the block might not be the latest assigned values when a computational interrupt occurs. Specifying ORDER impedes optimizing the allocation of registers but guarantees that all values of variables are reset in the block, thereby immediately assigning values to their storage locations.

---

## Program Constructs that Inhibit Optimization

The following sections describe source program constructs that can inhibit optimization.

## Global Optimization of Variables

The compiler considers 255 variables in the program for global optimization. It considers the remainder solely for local optimization.

The compiler considers explicitly declared variables for global optimization in preference to contextually declared variables, and then gives preference to contextually declared variables over implicitly declared variables. The highest preference is given to those variables declared in the final DECLARE statements in the outermost block.

If your program contains more than 255 variables, you can benefit most from the global optimization of arithmetic variables—particularly do-loop control variables and subscripting variables. You will gain little or no benefit from the optimization of string variables or program control data.

You should declare arithmetic variables in the final DECLARE statements in the outermost block rather than implicitly. You can benefit further if you eliminate declared but unreferenced variables from the program.

## ORDER and REORDER Options

ORDER and REORDER are optimization options specified for a procedure or begin-block in a PROCEDURE or BEGIN statement.

ORDER is the default for external procedures. The default for internal blocks is to inherit ORDER or REORDER from the containing block.

### ORDER Option

Specify the ORDER option for a procedure or begin-block if you must be sure that only the most recently assigned values of variables modified in the block are available for ON-units, which are entered because of computational conditions raised during the execution of statements and expressions in the block.

In a block with the ORDER option specified, the compiler might eliminate common expressions, causing fewer computational conditions to be raised during execution of the block than if common expressions had not been eliminated. But if a computational condition is raised during execution of an ORDER block, the values of variables in statements that precede the condition are the most recent values assigned when an ON-unit refers to them for the condition.

You can use other forms of optimization in an ORDER block, except for forward or backward move-out of any expression that can raise a condition. Since it would be necessary to disable all the possible conditions that might be encountered, the use of ORDER virtually suppresses any move-out of statements or expressions from loops.

### REORDER Option

The REORDER option allows the compiler to generate optimized code to produce the result specified by the source program, when error-free execution takes place. Move-out is allowed for any invariant statements and expressions from inside a loop to a point in the source program, either preceding or following such a loop. Thus, the statement or expression is executed once only, either before or after the loop.

More efficient execution of loops can be achieved by maintaining, in registers, the values of variables that are subject to frequent modification during the execution of the loops. When error-free execution allows, values can be kept in registers. This dispenses with time-consuming load-and-store operations needed to reset the values of variables in their storage locations. If the latest value of a variable is required after a loop has been executed, the value is assigned to the storage location of the variable when control passes out of the loop.

You can more significantly optimize register allocation if you specify REORDER for the block. However, the values of variables that are reset in the block are not guaranteed to be the latest assigned values when a computational condition is raised, since the latest value of a variable can be present in a register but not in the storage location of the variable. Thus, any ON-unit entered for a computational condition must not refer to variables set in the reorder block. However, use of the built-in functions ONSOURCE and ONCHAR is still valid in this context.

A program is in error if during execution there is a computational or system action interrupt in a REORDER block followed by the use of a variable whose value is not guaranteed.

Because of the REORDER restrictions, the only way you can correct erroneous data is by using the ONSOURCE and ONCHAR pseudovariables for a CONVERSION ON-unit. Otherwise, you must either depend on the implicit action, which terminates execution of the program, or use the ON-unit to perform error recovery and to restart execution by obtaining fresh data for computation. The second approach should ensure that all valid data is processed, and that invalid data is noted, while still taking advantage of any possible optimization. For example:

```
ON OVERFLOW PUT DATA;
DO J = 1 TO M;
DO I = 1 TO N;
X(I,J) = Y(I) + Z(J) *L + SQRT(W);
P = I*J;
END;
END;
```

When the above statements appear in a reorder block, the source code compiled is interpreted as follows:

```
ON OVERFLOW PUT DATA;
TEMP1 = SQRT(W);
DO J = 1 TO M;
TEMP2 = J;
DO I = 1 TO N;
X(I,J) = Y(I) +Z(J)*L+TEMP1;
P=TEMP2;
TEMP2=TEMP2+J;
END;
END;
```

TEMP1 and TEMP2 are temporary variables created to hold the values of expressions moved backward out of the loops, and the statement  $P=I*J$  can be simplified to  $P=N*M$ . If the OVERFLOW condition is raised, the values of the variables used in the loops cannot be guaranteed to be the most recent values

assigned before the condition was raised, since the current values can be held in registers, and not in the storage location to which the ON-unit must refer.

Although this example does not show it, the subscript calculations for X, Y, and Z are also optimized.

## Common Expression Elimination

Common expression elimination is inhibited by:

- The use in expressions of variables whose values can be reset in either an input/output or computational ON-unit.
- If a based variable is, at some point in the program, overlaid on top of a variable used in the common expression, assigning a new value to the based variable in between the two occurrences of the common expression, inhibits optimization.

For instance, the common expression  $X+Z$ , in the following example, is not eliminated because the based variable A which, earlier in the program, is overlaid on the variable X, is assigned a value in between the two occurrences of  $X+Z$ .

```
DCL A BASED(P);
P=ADDR(X);
.
.
.
P=ADDR(Y);
.
.
.
B=X+Z;
P->A=2;
C=X+Z;
```

- The use of aliased variables. An aliased variable is any variable whose value can be modified by references to names other than its own name. Examples are variables with the DEFINED attribute, variables used as the base for defined variables, parameters, arguments, and based variables.

Variables whose addresses are known to an external procedure by means of pointers that are either external or used as arguments are also assumed to be aliased variables.

The effect of an aliased variable does not completely prevent common expression elimination, but inhibits it slightly. For all aliased variables, the compiler builds a list of all variables that could possibly reference the aliased variable. The list is the same for each member of the list, and in a given program there can be many such lists.

When an expression containing an aliased variable is checked for its use as a common expression, the possible flow paths along which related common expressions could occur are also searched for assignments. The search is not only for the variable referenced in the expression, but also for all the members of the alias list to which that variable belongs. If the program contains an external pointer variable, it is assumed that this pointer could be set to all variables whose addresses are known to external procedures; that is, all external variables, all arguments passed to external procedures, and all variables whose addresses could be assigned to the external pointer. Thus,

variables addressed by the external pointer, or by any other pointer that has a value assigned to it from the external pointer, are assumed to belong to the same alias list as the external variables.

- The form of an expression. If the expression  $B+C$  could be treated as a common expression, the compiler would not be able to detect it as a common expression in the following statement:

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

The compiler processes the expression  $A+B+C$  from left to right. Consequently, it only recognizes the expressions  $A+B$  and  $(A+B)+C$ . However, by coding the expression  $D=A+(B+C)$ , you can ensure that it is recognized.

- The scope of a common expression. In order to determine the presence of common expressions, the program is analyzed and the existence of flow units is determined. A *flow unit* is a unit of object code that can only be entered at the first instruction and left at the last instruction. A flow unit can contain several PL/I source statements; conversely, a single PL/I source statement can comprise several flow units. Common expressions are recognized across individual flow units. However, if the program flow paths between flow units are complex, the recognition of common expressions is inhibited across flow units.

Common expression elimination is assisted by these points:

- Variables in expressions should not be external, associated with external pointers, or arguments to ADDR built-in functions.
- The source program should not contain external procedures, external label variables, or label constants known to external procedures.
- Variables in expressions should not be set or accessed in ON-units, if possible.
- Expressions to be commoned or transferred must be arithmetic (for example,  $A+B$ ) or string (for example,  $E||F$  or  $STRING(G)$ ) rather than compiler generated.

The type of source program construct discussed below could cause common expression elimination to be carried out when it should not be.

A PL/I block can access any element of an array or structure variable if the variable is within the block's name scope, or if it has been passed to the block. When using BASED or DEFINED variables, or pointer arithmetic under the control of the LANGLVL(SPROG) compile-time option, be careful not to access any element that has not been passed to the block, and whose containing structure or array has not been passed to the block and is not within the block's name scope. Any such attempt is invalid and might lead to unpredictable results.

In the following example, procedure X passes only the address of element A.C to procedure Y. The first assignment statement in procedure Y makes a valid change to A.C. However, other statements in procedure Y are invalid because they attempt to change the values of A.B and A.D, which procedure Y cannot legally access.

Because neither the containing structure A nor its elements B, D, and E are passed to procedure Y, elements B, D, and E are not aliased for use by procedure Y. Therefore, the compiler cannot detect that their values might change in procedure Y, so it performs common expression elimination on the expression "B + D + E" in

procedure X. Changing the values of A.B and A.D in procedure Y would then cause unpredictable results.

The GOTO statement in procedure Y is another error. It causes a flow of control change between blocks that is hidden from the compiler because neither A.F nor its containing variable A are passed to procedure Y. This invalid change in the flow of control can also cause unpredictable results.

```

X: PROCEDURE OPTIONS(MAIN);
  DECLARE Y ENTRY(POINTER) EXTERNAL;
  DECLARE
    1 A,
    2 B FIXED BIN(31) INIT(1),
    2 C FIXED BIN(31) INIT(2),
    2 D FIXED BIN(31) INIT(3),
    2 E FIXED BIN(31) INIT(4),
    2 F LABEL VARIABLE INIT(L1);
  N = B + D + E;
  CALL Y(ADDR(C));
L1: M = B + D + E;
  END X;
Y: PROCEDURE (P);
  DECLARE
    (P,Q) POINTER,
    XX FIXED BIN(31) BASED;
  DECLARE
    1 AA BASED,
    2 CC FIXED BIN(31),
    2 DD FIXED BIN(31),
    2 EE FIXED BIN(31),
    2 FF LABEL VARIABLE;
  P->XX = 17; /* valid change to A.C */
  Q = P - 4; /* invalid */
  Q->XX = 13; /* invalid change to A.B */
  P->DD = 11; /* invalid change to A.D */
  GOTO P->FF; /* invalid flow to label L1 */
  END Y;

```

## Condition Handling for Programs with Common Expression Elimination

The order of most operations in each PL/I statement depends on the priority of the operators involved. However, for sub-expressions whose results form the operands of operators of lower priority, the order of evaluation is not defined beyond the rule that an operand must be fully evaluated before its value can be used in another operation. These operands include subscript expressions, locator qualifier expressions, and function references. Therefore, ON-units associated with conditions raised during the evaluation of such sub-expressions can be entered in an unpredictable order. Consequently, an expression might have several possible values, according to the order of, and action taken by, the ON-units that are entered. When a computational ON-unit is entered:

- The values of all variables set by the execution of previous statements are guaranteed to be the latest values assigned to the variables, and can be used by the ON-unit. For instance the PUT DATA statement can be used to record the values of all variables on entry to an ON-unit.

- The value of any variable set in an ON-unit resulting from a computational interrupt is guaranteed to be the latest value assigned to the variable, for any part of the program.

Where there is a possibility that variables might be modified as the result of a computational interrupt, either in the associated ON-unit, or as the result of the execution of a branch from the ON-unit, common expression elimination is inhibited. For example:

```
ON ZERODIVIDE B,C=1;
.
.
.
X=A*B+B/C;
Y=A*B+D;
```

The compiler normally attempts to eliminate the reevaluation of the subexpression  $A*B$  in the second assignment statement. However, in this example, if the ZERODIVIDE condition is raised during the evaluation of  $B/C$ , the two values for  $A*B$  would be different. Common expression elimination is inhibited to allow for this possibility.

The above discussion applies only when the optimization option ORDER is specified or defaulted. If you do not require the guarantees described above, you can specify the optimization option REORDER. In this case, common expression elimination is not inhibited.

## Transfer of Invariant Expressions

Transfer of invariant expressions out of loops is inhibited by:

- ORDER specified for the block. However, transfer is not entirely prevented by the ORDER option. It is only inhibited for operations that can raise computational conditions. Such operations do not include array subscript manipulation where the subscripts are calculated with logical arithmetic which cannot raise OVERFLOW.
- The use of variables whose values can be set or used by input or output statements.
- The use of variables whose values can be set in input/output or computational ON-units, or which are aliased variables.
- A complicated program flow, involving external procedures, external label variables, and label constants.
- The use of a WHILE option in a repetitive DO statement. An invariant expression can be moved out of a do-group only if it appears in a statement that would have been executed during every repetition of the do-group. The appearance of a WHILE option in the DO statement of a repetitive do-group effectively causes each statement in the do-group to be considered "not always executed," since it might prevent the do-group from being executed at all. (You could, instead, have an equivalent do-group using an UNTIL option.)
- The appearance in an expression of a variant term or sub-expression preceding an invariant term or sub-expression. For example, assume that V is variant, and that NV1 and NV2 are invariant, in a loop containing the following statement:

```
X = V * NV1 * NV2;
```

The appearance of V preceding the sub-expression NV1\*NV2 prevents the movement of the evaluation of NV1\*NV2 out of the loop. (You could, instead, write X = NV1 \* NV2 \* V; getting the same result while allowing the sub-expression to be moved out of the loop.)

- The appearance in an expression of a constant that is not of the same data type as subsequent invariant elements in the expression. For example, assume that NV1 and NV2 are declared FLOAT DECIMAL, and are invariant in a loop containing the following statement:

```
X = 100 * NV1 * NV2;
```

The constant 100 has the attributes FIXED DECIMAL. For technical reasons beyond the scope of this discussion, this mismatch of attributes prevents the movement of the entire expression. (You could, instead, write the constant as 100E0, which has the attributes FLOAT DECIMAL.)

You can assist transfer by specifying REORDER for the block.

## Redundant Expression Elimination

Redundant expression elimination is inhibited or assisted by the same factors as for transfer of invariant expressions, described above.

## Other Optimization Features

Optimized code can be generated for the following items:

- A do-loop control variable, except when its value can be modified either explicitly or by an ON-unit during execution of a do-loop.
- Do-loops that do not contain other do-loops. This applies only if the scope of the control variable extends beyond the block containing the do-loop, it is given a definite value after the do-loop and before the end of the block.
- Assignment of arrays or structures, unless noncontiguous storage is used.
- Array initialization where the same value is assigned to each element, unless the array occupies noncontiguous storage.
- In-line conversions, unless they involve complicated picture or character-to-arithmetic conversions.
- In-line code for the string built-in functions SUBSTR and INDEX, unless the on-conditions STRINGSIZE or STRINGRANGE are enabled.
- Register allocation and addressing schemes, unless the program flow is complicated by use of external procedures, external label variables, or label constants known to external procedures. Optimized register usage is also inhibited by the use of aliased variables and variables that are referenced or set in an ON-unit.

---

## Assignments and Initialization

When a variable is accessed, it is assumed to have a value which was previously assigned to it, and which is consistent with the attributes of the variable. If this assumption is incorrect, the program either proceeds with incorrect data or raises the ERROR condition. Invalid values can result from failure to initialize the variable, or it can occur as a result of the variable being set in one of the following ways:

- By the use of the UNSPEC pseudovalue
- By record-oriented input
- By overlay defining a picture on a character string, with subsequent assignment to the character string and then access to the picture
- By passing as an argument an incompatible value to an external procedure, without matching the attributes of the parameter by an appropriate parameter-descriptor list
- By assignment to a based variable with different attributes, but at the same location.

Failure to initialize a variable results in the variable having an unpredictable value at run time. Do not assume this value to be zero.

Failure to initialize a subscript can be detected by enabling SUBSCRIPTRANGE, when debugging the program (provided the uninitialized value does not lie within the range of the subscript).

Referencing a variable that has not been initialized can raise a condition. For example:

```
DCL A(10) FIXED;  
A(1)=10;  
PUT LIST (A);
```

The array should be initialized before the assignment statement, thus:

```
A=0;
```

---

## Notes about Data Elements

- Take special care to make structures match when you intend to move data from one structure to another. This avoids conversion and also allows the structure to be moved as a unit instead of element by element.
- Use pictured data rather than character data if possible. For example, if a piece of input data contains 3 decimal digits, and neither ONSOURCE nor ONCHAR is used to correct invalid data:

```
DCL EXTREP CHARACTER(3),  
      INTREP FIXED DECIMAL (5,0);
```

```
ON CONVERSION GOTO ERR;  
INTREP = EXTREP;
```

is less efficient than:

```

DCL EXTREP CHARACTER(3),
    PICREP PIC '999' DEF EXTREP,
    INTREP FIXED DECIMAL (5,0);

IF VERIFY(EXTREP,'0123456789')
    ¬= 0 THEN GOTO ERR;
INTREP = PICREP;

```

- Declare the FIXED BINARY (31,0) attribute for internal switches and counters, and variables used as array subscripts.
- Exercise care in specifying the precision and scale factor of variables that are used in expressions. Using variables that have different scale factors in an expression can generate additional object code that creates intermediate results.
- You should, if possible, specify and test bit strings as multiples of eight bits. However, you should specify bit strings used as logical switches according to the number of switches required. In the following examples, (a) is preferable to (b), and (b) is preferable to (c):

#### Example 1:

##### Single switches

- ```

(a) DCL SW BIT(1) INIT('1'B);
    IF SW THEN...

(b) DCL SW BIT(8) INIT('1'B);
    IF SW THEN...

(c) DCL SW BIT(8) INIT ('1'B);
    IF SW = '10000000'B THEN...

```

#### Example 2:

##### Multiple switches

- ```

(a) DCL B BIT(3);
    B = '111'B;
    .
    .
    .
    IF B = '111'B THEN DO;

(b) DCL B BIT(8);
    B = '11100000'B;
    .
    .
    .
    IF B = '11100000'B THEN DO;

(c) DCL (SW1,SW2,SW3) BIT(1);
    SW1, SW2, SW3 = '1'B;
    .
    .
    .
    IF SW1&SW2&SW3 THEN DO;

```

- If bit-string data whose length is not a multiple of 8 is to be held in structures, you should declare such structures ALIGNED.

- Varying-length strings are generally less efficient than fixed-length strings.

Fixed-length strings are not efficient if their lengths are not known at compile time, as in the following example:

```
DCL A CHAR(N);
```

- When storage conservation is important, use the UNALIGNED attribute to obtain denser packing of data, with a minimum of padding.
- The precision of complex expressions follows the rules for expression evaluation. For example, the precision of  $1 + 1i$  is (2,0).
- Do not use control characters in a source program for other than their defined meanings, such as delimiting a graphic string. (Control characters are those characters whose hexadecimal value is in the range from 00 to 3F, or is FF.) Although such use is not invalid, it can make it impossible for the source program to be processed by products (both hardware and software) that implement functions for the control characters.

For example, you might write a program to edit data for a device that recognizes hexadecimal FF embedded in the data as a control character meant to suppress display. Then, if you write:

```
DCL SUPPRESS CHAR(1) STATIC INIT(' ');
```

where the INIT value is hexadecimal FF, your program is not displayed on that device, because the control character is recognized and suppresses all displaying after the INIT.

Or, suppose you are creating a record that requires 1-byte numeric fields and you choose to code the following:

```
DCL X01 CHAR(1) STATIC INIT(' '); /* HEXADECIMAL'01' */
```

Your program can fail if it is processed by a product that recognizes control characters.

To avoid these problems, represent control character values by using X character string constants. For example:

```
DCL SUPPRESS CHAR(1) INIT('FF'X);
```

```
DCL X01 CHAR(1) INIT('01'X);
```

Thus you have the values required for execution without having control characters in the source program, you are not dependent on the bit representation of characters, and the values used are “visible” in your source program.

- In the case of unaligned fields, code is sometimes generated to move data to aligned fields. Thus, if you correctly align your data and specify the ALIGNED attribute you will avoid extraneous move instructions and improve execution speed for aligned data.
- If you test a read-only bit field in protected storage such that modification of that field would result in a protection exception, certain coding should be avoided. For example:

```
Dcl bitfld bit(64) based(p); /* Multiple of 8 bits */
If bitfld then ... /* NC hardware instruction */
```

where bitfld is a whole number of bytes and is byte aligned. To test whether bitfld is zeros, the compiler attempts to generate efficient code and so executes

|  
| an NC hardware instruction, which ANDs the bitfld to itself and sets the  
| appropriate hardware condition code. The field is unchanged by this operation.  
| However, if the field is not modifiable (read-only at a hardware level), a  
| protection exception will occur.

|  
| To avoid such a protection exception, you should explicitly test for a non=zero  
| bit string, as shown in the following example:

|  
| Dcl bitfld bit(64) based(p);  
| Dcl b0 bit(0) static;  
| If bitfld=b0 then ...

---

## Notes about Expressions and References

- Do not use operator expressions, variables, or function references where constants can be substituted.

For example:

```
DECLARE A(8);
```

is more efficient than

```
DECLARE A(5+3);
```

and

```
VERIFY(DATA, '0123456789')...;
```

is more efficient than

```
DCL NUMBERS CHAR(10) STATIC INIT('0123456789');  
VERIFY(DATA, NUMBERS)...;
```

The preprocessor is very effective in allowing the source program to be symbolic, with expression evaluation and constant substitution taking place at compile time. The examples above could be:

```
% DCL DIMA FIXED;  
% DIMA = 5+...;  
DCL A(DIMA);
```

and

```
% DCL NUMBERS CHAR;  
% NUMBERS='''0123456789''';  
VERIFY(DATA, NUMBERS)...;
```

- Use additional variables to avoid conversions. For example, consider a program in which a character variable is to be regularly incremented by 1:

```
DECLARE CTLNO CHARACTER(8);  
CTLNO = CTLNO + 1;
```



```
B=A(1,1);
A=A/B;
```

or:

```
DCL A(10,20),
     B(10,20);
B=A/A(1,1);
```

- Array multiplication does not affect matrix multiplication. For example:

```
DCL (A,B,C) (10,10);
A=B*C;
```

is equivalent to:

```
DCL (A,B,C) (10,10);
DO I=1 TO 10;
DO J=1 TO 10;
A(I,J)=B(I,J)*C(I,J);
END; END;
```

Table 41. Conditions under Which String Operations Are Handled In-Line

String operation	Comments and conditions		
	Source	Target	Comments
Assign	Nonadjustable, ALIGNED, fixed-length bit string	Nonadjustable, ALIGNED, bit string	No length restriction if OPTIMIZE(TIME) is specified; otherwise maximum length of 8192 bits
	Adjustable or VARYING, ALIGNED bit string	Nonadjustable, ALIGNED bit string of length $\leq 2048$	Only if OPTIMIZE(TIME) is specified
	Nonadjustable, fixed-length character string	Nonadjustable character string	
	Adjustable or VARYING character string	Nonadjustable character string of length $\leq 256$	
'and', 'not', 'or'	As for bit string assignments but no adjustable or varying-length operands are handled		
Compare	As for string assignment with the two operands taking the roles of source and target, but no adjustable or varying-length operands are handled		
Concatenate	As for string assignments, but no adjustable or varying-length source strings are handled		
STRING built-in function	When the argument is an element variable, or a nonadjustable array or structure variable in connected storage		
<b>Note:</b> If the target is fixed-length, the maximum length is the target length. If the target is VARYING, the maximum length is the lesser of the operand lengths.			

---

## Notes about Data Conversion

The data conversions performed in-line are shown in Table 42 on page 330. A conversion outside the range or condition given is performed by a library call.

Not all the picture characters available can be used in a picture involved in an in-line arithmetic conversion. The only ones allowed are:

- V and 9
- Drifting or nondrifting characters \$ S and +
- Zero suppression characters Z and \*
- Punctuation characters , . / and B.

For in-line conversions, pictures with this subset of characters are divided into three types:

### Picture type 1:

Pictures of all 9s with an optional V and a leading or trailing sign. For example:

'99V999', '99', 'S99V9', '99V+', '\$999'

### Picture type 2:

Pictures with zero suppression characters and optional punctuation characters and a sign character. Also, type 1 pictures with punctuation characters. For example:

'ZZZ', '\*\*/\*\*9', 'ZZ9V.99', '+ZZ.ZZZ', '\$///99', '9.9'

### Picture type 3:

Pictures with drifting strings and optional punctuation characters and a sign character. For example:

'\$\$\$\$', '-,--9', 'S/SS/S9', '+++9V.9', '\$\$\$9-'

Sometimes a picture conversion is not performed in-line even though the picture is one of the above types. This might be because:

- SIZE is enabled and could be raised.
- There is no correspondence between the digit positions in the source and target. For example, DECIMAL (6,8) or DECIMAL (5,-3) to PIC '999V99' will not be performed in-line.
- The picture can have certain characteristics that make it difficult to handle in-line. For example:

- Punctuation between a drifting Z or a drifting \* and the first 9 is not preceded by a V. For example:

'ZZ.99'

- Drifting or zero suppression characters to the right of the decimal point. For example:

'ZZV.ZZ' or '++V++'

Table 42 (Page 1 of 2). Implicit Data Conversion Performed In-Line

Conversion		Comments and conditions
Source	Target	
FIXED BINARY	FIXED BINARY	None.
	FIXED DECIMAL	None.
	FLOAT	Long or short FLOAT target.
	Bit string	String must be fixed-length, ALIGNED, and with length less than or equal to 2048. STRINGSIZE condition must be disabled.
	Character string or Numeric picture	Via FIXED DECIMAL. String must be fixed-length with length less than or equal to 256 and STRINGSIZE disabled. Picture types 1, 2, or 3 when SIZE disabled.
FIXED DECIMAL	FIXED BINARY	None.
	FIXED DECIMAL	None.
	FLOAT	If q1+p1 less than or equal to 75. Long or short-FLOAT target.
	Bit string	String must be fixed-length, ALIGNED, and with length less than or equal to 2048. STRINGSIZE condition must be disabled.
	Character string	If precision = scale factor, it must be even. String must be fixed-length and length less than or equal to 256. STRINGSIZE must be disabled.
FLOAT (Long or Short)	Numeric picture	Picture types 1, 2, and 3.
	FIXED BINARY	None.
	FIXED DECIMAL	Scale factor less than 80.
	FLOAT	Source and target can be single or double length.
	Bit string	String must be fixed-length, ALIGNED, and with length less than or equal to 2048. STRINGSIZE condition must be disabled.
Bit string	FIXED BINARY	Source string must be fixed-length, ALIGNED, and with length less than or equal to 32.
	FIXED DECIMAL and FLOAT	Source must be fixed-length, ALIGNED, and with length less than 32.
	Character string	Source must be fixed-length, ALIGNED, and length 1 only.
Character	Bit string	None.
	FIXED DECIMAL	Source length 1 only. CONVERSION condition must be disabled.
	FLOAT	None.
	FIXED BINARY	None.
Character picture	Character string	String must be fixed-length with length less or equal to 256.
	Character picture	Pictures must be identical.

Table 42 (Page 2 of 2). Implicit Data Conversion Performed In-Line

Conversion		Comments and conditions
Source	Target	
Numeric picture type 1 and 2	FIXED BINARY	Via FIXED DECIMAL. SIZE condition disabled.
	FIXED DECIMAL	Type 2 picture without * or embedded punctuation. SIZE condition disabled.
	FLOAT	Via FIXED DECIMAL. Size condition disabled.
	Numeric picture	Picture types 1, 2, or 3. SIZE condition disabled.
Locator	Locator	None.

## Notes about Program Organization

- Although you can write a program as a single external procedure, it is often desirable to design and write the program as a number of smaller external procedures, or modules. In PL/I, the basic units of modularity are the procedure and the begin-block.

Some of the advantages of modular programming are:

- Program size affects the time and space required for compilation. Generally, compilation time increases more than linearly with program size, especially if the compiler has to spill text onto auxiliary storage. Also, the process of adding code to a program and then recompiling it leads to wasteful multiple-compilation of existing text. Modular programming eliminates the above possibilities.
- If you design a procedure to perform a single function it needs to contain only the data areas for that function. Because of the nature of AUTOMATIC storage, there is less danger of corruption of data areas for other functions.
- If you design a procedure to perform a single function, you can more easily replace it with a different version. Also, you can use the same procedure in several different applications.
- Storage allocation for all the automatic variables in a procedure occurs when the procedure is invoked at any of its entry points. If you reduce the number of functions performed by a procedure, you can often reduce the number of variables declared in the procedure. This in turn can reduce the overall demand for storage for automatic variables.

More important (from the efficient programming viewpoint) are the following considerations:

- The compiler has a limitation on the number of variables that it can consider for global optimization. (The number of variables does not affect other forms of optimization.)
- The compiler has a limitation on the number of flow units that it can consider for flow analysis and, subsequently, for global optimization.
- If the static CSECT or the DSA exceeds 4096 bytes in size, the compiler has to generate additional code to address the more remote storage.

- If the object code for a procedure exceeds 4096 bytes in size, the compiler might have to repeatedly reset base registers.

Extra invocation of procedures increases run time, but the use of modular programming often offsets the increase, because the additional optimization can cause significantly fewer instructions to be executed.

- Avoid *unnecessary* program segmentation and block structure. This includes all procedures, ON-units, and begin-blocks that need activation and termination. The initialization and program management for these carry an overhead.

You should assess this recommendation in conjunction with the notes on modular programming given earlier in this section.

- The procedure given initial control at run time must have the OPTIONS(MAIN) attribute. If more than one procedure has the MAIN option, the first one encountered by the linkage editor receives control.
- Under the compiler, attempting the recursive use of a procedure that was not given the RECURSIVE attribute can result in the ERROR condition after exit from the procedure. This occurs if reference is made to automatic data from an earlier invocation of the procedure.

---

## Notes about Recognition of Names

Separate external declarations for the same variable must not specify conflicting attributes (CONTROLLED EXTERNAL variables are an exception), either explicitly or by default. If this occurs, the compiler is not able to detect the conflict.

---

## Notes about Storage Control

- Using self-defining data (the REFER option) enables data to be held in a compact form; it can, however, produce less-than-optimum object code. For example, with the structure:

```
DCL 1 STR BASED(P),  
  2 N,  
  2 BEFORE,  
  2 ADJUST (NMAX REFER(N)),  
    3 BELOW,  
  2 AFTER;
```

a reference to BEFORE requires one instruction to address the variable, whereas a reference to AFTER or BELOW requires approximately 18 instructions, plus a call to a resident library module.

You can organize a self-defining structure more efficiently by ensuring that the members whose lengths are known appear before any adjustable members, and that the most frequently used adjustable members appear before those that are less frequently used. The previous example could thus be changed to:

```
DCL 1 STR BASED(P),  
  2 N,  
  2 BEFORE,  
  2 AFTER,  
  2 ADJUST (NMAX REFER(N)),  
    3 BELOW;
```

- PL/I does not allow the use of the INITIAL attribute for arrays that have the attributes STATIC and LABEL, because the environmental information for the array does not exist until run time.

However, when compiling procedures containing STATIC LABEL arrays, you might improve performance by specifying the INITIAL attribute, provided that the number of elements in the array is less than 512. What happens under these conditions is:

- The compiler diagnoses the invalid language (STATIC, LABEL, and INITIAL) and produces message IEL0580I, but it accepts the combination of attributes.
  - If OPT(TIME) is in effect, the compiler checks GO TO statements that refer to STATIC LABEL variables to see whether the value of the label variable is a label constant in the same block as the GO TO statement. If the value is a label constant in the same block, the compiler replaces the normal interpretative code with direct branching code. If it is not, the compiler produces message IEL0918I and the interpretative code remains unchanged.
  - If OPT(0) is in effect, or if message IEL0918I is produced, or if the array is larger than 512 elements, execution is liable to terminate with an interrupt, or go into a loop, or produce other unpredictable results.
- AUTOMATIC variables are allocated at entry to a block; sequences of the following type allocate B with the value that N had on entry to the block (not necessarily 4):

```
A: PROC;
   N=4;
   DCL B(N) FIXED;
   .
   .
   .
   END;
```

- If you specify the INITIAL attribute for an external noncontrolled variable, you must specify it, with the same value, on all the declarations for that variable. An exception to this rule is that an INITIAL attribute specified for an external variable in a compiled procedure need not be repeated elsewhere.
- String lengths, area sizes, and array bounds of controlled variables can be recomputed at the time of an ALLOCATE statement. As a result, even if the declaration asserts that a structure element (for example, X) is CHARACTER(16) or BIT(1), PL/I treats the declaration as CHARACTER(\*) or BIT(\*). Thus, library subroutines manipulate these strings. The compiler allocates and releases the temporary space needed for calculations (for instance, concatenation) using these strings each time the calculation is performed, since the compiler does not know the true size of the temporary space.

An alternate way of declaring character strings of known length is to declare PICTURE '(16)X' to hold a 16-character string. Because the number "16" is not subject to change at ALLOCATE time, better code results.

- A variable with adjustable size that is defined on an external variable cannot be passed as an argument to a procedure.

---

## Notes about Statements

- If a GOTO statement references a label variable, it is more efficient if the label constants that are the values of the label variable appear in the same block as the GOTO statement.
- When storage conservation is important, avoid the use of iterative do-groups with multiple specifications. The following is inefficient in terms of storage requirements:

```
DO I = 1,3,-5,15,31;  
.  
.  
.  
END;
```

- A repetitive do-group is not executed if the terminating condition is satisfied at initialization:

```
I=6;  
DO J=I TO 4;  
  X=X+J;  
END;
```

This group does not alter X, since BY 1 is implied. Iterations can step backwards, and if BY -1 was specified, three iterations would take place.

- Expressions in a DO statement are assigned to temporaries with the same characteristics as the expression, not the variable. For example:

```
DCL A DECIMAL FIXED(5,0);  
A=10;  
DO I=1 TO A/2;  
.  
.  
.  
END;
```

This loop is not executed, because A/2 has decimal precision (15,10), which, on conversion to binary (for comparison with I), becomes binary (31,34).

Five iterations result if the DO statement is replaced by:

```
ITEMP=A/2;  
DO I=1 TO ITEMP;
```

or:

```
DO I=1 TO PREC(A/2,6,1)
```

- Upper and lower bounds of iterative do-groups are computed only once, even if the variables involved are reassigned within the group. This applies also to the BY expression.

Any new values assigned to the variables involved take effect only if the do-group is started again.

- In a do-group with both a control variable and a WHILE option, the evaluation and testing of the WHILE expression is carried out only after determination (from the value of the control variable) that iteration can be performed. For example, the following group is executed at most once:

```

DO I=1 WHILE(X>Y);
.
.
.
END;

```

- If the control variable is used as a subscript within the do-group, ensure that the variable does not run beyond the bounds of the array dimension. For instance:

```

DECLARE A(10);
DO I = 1 TO N;
.
.
.
A(I) = X;
.
.
.
END;

```

If N is greater than 10, the assignment statement can overwrite data beyond the storage allocated to the array A. A bounds error can be difficult to find, particularly if the overwritten storage happens to contain object code. You can detect the error by enabling SUBSCRIPTRANGE.

For CMPAT(V2), you should explicitly declare the control variable as FIXED BIN (31,0), because the default is FIXED BIN (15,0) which cannot hold fullword subscripts.

- If a Type 2 DO specification includes both the WHEN and UNTIL options, the DO statement provides for repetitive execution, as defined by the following:

```

LABEL: DO WHILE (expression1)
        UNTIL (expression2)
        statement-1
.
.
.
        statement-n
        END;
NEXT:  statement /* STATEMENT FOLLOWING THE DO GROUP */

```

The above is exactly equivalent to the following expansion:

```

LABEL: IF (expression 1) THEN; ELSE
        GO TO NEXT;
        statement-1
.
.
.
        statement-n
LABEL2: IF (expression2) THEN; ELSE
        GO TO LABEL;
NEXT:  statement /* STATEMENT FOLLOWING THE DO GROUP */

```

The statement GO TO LABEL; replaces the IF statement at label LABEL2 in the expansion if the UNTIL option is omitted.

A null statement replaces the IF statement at label LABEL. Note that, if the WHILE option is omitted, statements 1 through n are executed at least once.

- If the Type 3 DO specification includes the TO and BY options, the action of the do-group is defined by the following:

```

LABEL:  DO variable=
        expression1
        TO expression2
        BY expression3
        WHILE (expression4)
        UNTIL(expression5);
        statement-1
        .
        .
        .
        statement-m
LABEL1:  END;
NEXT:   statement

```

For a variable that is not a pseudovvariable, the above action of the do-group definition is exactly equivalent to the following expansion, in which 'p' is a compiler-created pointer, 'v' is a compiler-created based variable based on a 'p' and with the same attributes as 'variable,' and 'en' ( $n=1, 2, \text{ or } 3$ ) is a compiler-created variable: For example:

```

LABEL:  p=ADDR(variable);
        e1=expression1;
        e2=expression2;
        e3=expression3;
        v=e1;
LABEL2: IF (e3>=0)&(v>e2) |
        (e3<0)&(v<e2)
        THEN GO TO NEXT;
        IF (expression4) THEN;
        ELSE GO TO NEXT;
        statement-1
        .
        .
        .
        statement-m
LABEL1: IF (expression5) THEN
        GO TO NEXT;
LABEL3: v=v+e3;
        GO TO LABEL2;
NEXT:   statement

```

If the specification includes the REPEAT option, the action of the do-group is defined by the following:

```
LABEL: DO variable=  
        expression1  
        REPEAT expression6  
        WHILE (expression4)  
        UNTIL(expression5);  
        statement-1  
        .  
        .  
        .  
        statement-m  
LABEL1: END;  
NEXT:  statement
```

For a variable that is not a pseudovisible, the above action of the do-group definition is exactly equivalent to the following expansion:

```
LABEL: p=ADDR(variable);  
        e1=expression1;  
        v=e1;  
LABEL2: ;  
        IF (expression4) THEN;  
        ELSE GO TO NEXT;  
        statement-1  
        .  
        .  
        .  
        statement-m  
LABEL1: IF (expression5) THEN  
        GO TO NEXT;  
LABEL3: v=expression6;  
        GO TO LABEL2;  
NEXT:  statement
```

Additional rules for the above expansions follow:

- The above expansion only shows the result of one specification. If the DO statement contains more than one specification, the statement labeled NEXT is the first statement in the expansion for the next specification. The second expansion is analogous to the first expansion in every respect. Note, however, that statements 1 through m are not actually duplicated in the program.
- Omitting the WHILE clause also omits the IF statement immediately preceding statement-1 in each of the expansions.
- Omitting the UNTIL clause also omits the IF statement immediately following statement-m in each of the expansions.

---

## Notes about Subroutines and Functions

- You should consider the precision of decimal constants when such constants are passed. For example:

```
CALL ALPHA(6);

ALPHA:PROCEDURE(X);
    DCL X FIXED DECIMAL;
    END;
```

If ALPHA is an external procedure, the above example is incorrect because X is given default precision (5,0), while the decimal constant, 6, is passed with precision (1,0).

- When a procedure contains more than one entry point, with different parameter lists on each entry, do not make references to parameters other than those associated with the point at which control entered the procedure. For example:

```
A:PROCEDURE(P,Q);
    P=Q+8; RETURN;
B: ENTRY(R,S);
    R=P+S;      /* THE REFERENCE TO P IS AN ERROR */
    END;
```

When an EXTERNAL ENTRY is declared with no parameter descriptor list or with no parameters, no matching between parameters and arguments will be done. Therefore, if any arguments are specified in a CALL to such an entry point, no diagnostic message will be issued.

If one of the following examples is used:

```
DECLARE X ENTRY EXTERNAL;
```

or

```
DECLARE X ENTRY() EXTERNAL;
```

any corresponding CALL statement, for example:

```
CALL X(parameter);
```

will not be diagnosed, although it might be an error.

---

## Notes about Built-In Functions and Pseudovariables

For efficiency, do not refer to the DATE built-in function more than once in a run. For example, instead of:

```
PAGEA= TITLEA||DATE;
PAGEB= TITLEB||DATE;
```

it is more efficient to write:

```
DTE=DATE;
PAGEA= TITLEA||DTE;
PAGEB= TITLEB||DTE;
```

Table 43 on page 339 shows the conditions under which string built-in functions are performed in-line.

Assignments made to a varying string by means of the SUBSTR pseudovisible do not set the length of the string. A varying string initially has an undefined length, so that if all assignments to the string are made using the SUBSTR pseudovisible, the string still has an undefined length and cannot be successfully assigned to another variable or written out.

When UNSPEC is used as a pseudovisible, the expression on the right is converted to a bit string. Consequently, the expression must not be invalid for such conversion. For example, if the expression is a character string containing characters other than 0 or 1, a conversion error will result.

If both operands of the MULTIPLY built-in function are FIXED DECIMAL and have precision greater than or equal to 14, unpredictable results can occur if SIZE is enabled.

If an array or structure, whose first element is an unaligned bit string that does not start on a byte boundary, is used as an argument in a call to a subroutine or a built-in function, the results are unpredictable.

*Table 43. Conditions under Which String Built-In Functions Are Handled In-Line*

String function	Comments and conditions
BIT	Always
BOOL	The third argument must be a constant. The first two arguments must satisfy the conditions for 'and', 'or', and 'not' operations in Table 41 on page 328.
CHAR	Always
HIGH	Always
INDEX	Second argument must be a nonadjustable character string less than 256 characters long.
LENGTH	Always
LOW	Always
REPEAT	Second argument must be constant.
SUBSTR	STRINGRANGE must be disabled.
TRANSLATE	First argument must be fixed-length; second and third arguments must be constant.
UNSPEC	Always
VERIFY	First argument must be fixed-length: <ul style="list-style-type: none"> <li>• If CHARACTER it must be less than or equal to 256 characters,</li> <li>• If BIT it must be ALIGNED, less than or equal to 2048 bits.</li> </ul> Second argument must be constant.

## Notes about Input and Output

- Allocate sufficient buffers to prevent the program from becoming I/O bound, and use blocked output records. However, consider the impact on other programs executing within the system.
- When creating or accessing a data set having fixed-length records, specify FS or FBS record format, whenever possible.

- When a number of data sets are accessed by a single statement, use of a file variable rather than the TITLE option can improve program efficiency by allowing a file for each data set to remain open for as long as the data set is required by the program. Using the TITLE option requires closing and reopening a file whenever the statement accesses a new data set.
- The OPEN statement is executed by library routines that are loaded dynamically at the time the OPEN statement is executed. Consequently, run time could be improved if you specify more than one file in the same OPEN statement, since the routines need be loaded only once, regardless of the number of files being opened. However, opening multiple files can temporarily require more internal storage than might otherwise be needed.

As with the OPEN statement, closing more than one file with a single CLOSE statement can save run time, but it can require the temporary use of more internal storage than would otherwise be needed.

---

## Notes about Record-Oriented Data Transmission

- If you declare a file DIRECT with the INDEXED option of the ENVIRONMENT attribute, you should also apply the ENVIRONMENT options INDEXAREA, NOWRITE, and ADDBUFF if possible.
- When you create or access a CONSECUTIVE data set, use file and record variable declarations that generate in-line code, if possible. Details of the declarations are given in Chapter 8, “Defining and Using Consecutive Data Sets” on page 129.
- Conversion of source keys for REGIONAL data sets can be avoided if the following special cases are observed:
  - For REGIONAL(1): When the source key is a fixed binary element variable or constant, use precision in the range (12,0) to (23,0).
  - For REGIONAL(2) and (3): When the source key is of the form (character expression||r), where r is a fixed binary element variable or constant, use precision in the range (12,0) to (23,0).
- Direct update of an INDEXED data set slows down if an input/output operation on the same file intervenes between a READ and a REWRITE for the same key. This can cause the REWRITE statement to issue an extra READ.
- A pointer set in READ SET or LOCATE SET is not valid beyond the next operation on the file, or beyond a CLOSE statement. In OUTPUT files, you can mix WRITE and LOCATE statements freely.
- When you need to rewrite a record that was read into a buffer (using a READ SET statement that specifies a SEQUENTIAL UPDATE file) and then updated, you can use the REWRITE statement without a FROM option. A REWRITE after a READ SET always rewrites the contents of the last buffer onto the data set.

For example:

```
3 READ FILE (F) SET (P);
.
.
.
5 P->R = S;
.
.
.
7 REWRITE FILE (F);
.
.
.
11 READ FILE (F) INTO (X);
.
.
.
15 REWRITE FILE (F);
.
.
.
19 REWRITE FILE (F) FROM (X);
```

Statement 7 rewrites a record updated in the buffer.

Statement 15 does not change the record on the data set at all.

Statement 19 raises the ERROR condition, since there is no preceding READ statement.

- There is one case where it is not possible to check for the KEY condition on a LOCATE statement until transmission of a record is attempted. This is when there is insufficient room in the specified region to output the record on a REGIONAL(3) V- or U-format file. Neither the record raising the condition nor the current record is transmitted.

If a LOCATE is the last I/O statement executed before the file closes, the record is not transmitted and the condition can be raised by the CLOSE statement.

---

## Notes about Stream-Oriented Data Transmission

- A common programming practice is to put the format list used by edit-directed input/output statements into a FORMAT statement. The FORMAT statement is then referenced from the input/output statement by means of the R-format item. For example:

```
DECLARE NAME CHARACTER(20), MANNO DEC FIXED(5,0);
.
.
.
PUT EDIT (MANNO,NAME) (R(OUTF));
.
.
.
OUTF:FORMAT(F(8),X(5),A(20));
```

Programming in this way reduces the level of optimization that the compiler is able to provide for edit-directed input/output. If the format list repeats in each input/output statement:

```
PUT EDIT (MANN0,NAME) (F(8),X(5),A(20));
```

the compiler is able to generate more efficient object code which calls fewer library modules, with a consequent saving in run time and load module size.

- In STREAM input/output, do as much as possible in each input/output statement. For example:

```
PUT EDIT ((A(I) DO I = 1 TO 10))(...;
```

is more efficient than:

```
DO I = 1 TO 10;
  PUT EDIT (A(I))(...
END;
```

- Use edit-directed input/output in preference to list- or data-directed input/output.
- Consider the use of overlay defining to simplify transmission to or from a character string structure. For example:

```
DCL 1 IN,
    2 TYPE CHAR(2),
    2 REC,
    3 A CHAR(5),
    3 B CHAR(7),
    3 C CHAR(66);
GET EDIT(IN) (A(2),A(5),A(7),A(66));
```

In the above example, each format-item/data-list-item pair is matched separately, code being generated for each matching operation. It would be more efficient to define a character string on the structure and apply the GET statement to the string:

```
DCL STRNG CHAR(80) DEF IN;
GET EDIT (STRNG) (A(80));
```

- When an edit-directed data list is exhausted, no further format items will be processed, even if the next format item does not require a matching data item. For example:

```
DCL A FIXED(5),
    B FIXED(5,2);
GET EDIT (A,B) (F(5),F(5,2),X(70));
```

The X(70) format item is not processed. To read a following card with data in the first ten columns only, the SKIP option can be used:

```
GET SKIP EDIT (A,B) (F(5), F(5,2));
```

- The number of data items represented by an array or structure name appearing in a data list is equal to the number of elements in the array or structure; thus if more than one format item appears in the format list, successive elements will be matched with successive format items.

For example:

```
DCL 1 A,  
    2 B CHAR(5),  
    2 C FIXED(5,2);  
.  
.  
.  
PUT EDIT (A) (A(5),F(5,2));
```

B is matched with the A(5) item, and C is matched with the F(5,2) item.

- If the ON statement:

```
ON ERROR PUT DATA;
```

is used in an outer block, you must remember that variables in inner blocks are not known and, therefore, are not dumped. You might wish, therefore, to repeat the ON-unit in all inner blocks during debugging.

- When you use PUT DATA without a data-list, every variable known at that point in the program is transmitted in data-directed output format to the specified file. Users of this facility, however, should note that:
  - Uninitialized FIXED DECIMAL data might raise the CONVERSION condition or the ERROR condition.
  - Unallocated CONTROLLED data causes arbitrary values to print and, in the case of FIXED DECIMAL, can raise the CONVERSION condition or the ERROR condition.
- An output file generated with PUT LIST contains a blank after the last value in each record. If the file is edited with an edit program that deletes trailing blanks, then values are no longer separated and the file cannot be processed by GET LIST statements.

---

## Notes about Picture Specification Characters

- In a PICTURE declaration, the V character indicates the scale factor, but does not in itself produce a decimal point on output. The point picture character produces a point on output, but is purely an editing character and does not indicate the scale factor. In a decimal constant, however, the point does indicate the scale factor. For example:

```
DCL A PIC'99.9',  
    B PIC'99V9',  
    C PIC'99.V9';  
A,C=45.6;  
B=7.8;  
PUT LIST (A,B,C);
```

This puts out the following values for A, B, and C, respectively:

```
04.5    078    45.6
```

If these values were now read back into the variables by a GET LIST statement, A, B, and C would be set to the following respective values:

```
004    780    45.6
```

If the PUT statement were then repeated, the result would be:

```
00.4    780    45.6
```

- Checking of a picture is performed only on assignment into the picture variable:

```
DCL A PIC'999999',
    B CHAR(6) DEF A,
    C CHAR(6);
B='ABCDEF';
C=A;      /* WILL NOT RAISE CONV CONDITION */
A=C;      /* WILL RAISE CONV */
```

Note also (A, B, C as declared above):

```
A=123456; /* A HAS VALUE 123456 */
          /* B HAS VALUE '123456' */
C=123456; /* C HAS VALUE 'bbb123' */
C=A;     /* C HAS VALUE '123456' */
```

---

## Notes about Condition Handling

- Even though a condition was explicitly disabled by means of a condition prefix, a lot of processing can be required if the condition is raised. For example, consider a random number generator in which the previous random number is multiplied by some factor and the effects of overflow are ignored:

```
DECLARE (RANDOM,FACTOR) FIXED BINARY(31,0);
(NOFOFL):RANDOM=RANDOM*FACTOR;
```

If the product of RANDOM and FACTOR cannot be held as FIXED BINARY(31,0), a hardware program-check interrupt occurs. This has to be processed by both the operating system interrupt handler and the PL/I error handler before it can be determined that the FIXEDOVERFLOW condition was disabled.

- If possible, avoid using ON-units for the FINISH condition. These increase the time taken to terminate a PL/I program, and can also cause loops, abends, or other unpredictable results.
- After debugging, disable any normally disabled conditions that were enabled for debugging purposes by removing the relevant prefixes, rather than by including NO-condition prefixes. For instance, disable the SIZE condition by removing the SIZE prefix, rather than by adding a NOSIZE prefix. The former method allows the compiler to eliminate code that checks for the condition, whereas the latter method necessitates the generation of extra code to prevent the checks being carried out.
- If the specified action is to apply when the named condition is raised by a given statement, the ON statement must be executed before that statement. The statements:

```
GET FILE (ACCTS) LIST (A,B,C);
ON TRANSMIT (ACCTS) GO TO TRERR;
```

result in the ERROR condition being raised in the event of a transmission error during the first GET operation, and the required branch is not taken (assuming that no previous ON statement applies). Furthermore, the ON statement is executed after each execution of the GET statement.

- At normal exit from an AREA ON-unit, the implicit action is to try again to make the allocation. As a result, the ON-unit is entered again, and an indefinite loop is created. To avoid this, you should modify the amount allocated in the ON-unit, for example, by using the EMPTY built-in function or by changing a pointer variable.
- Do not use ON-units to implement the program's logic; use them only to recover from truly exceptional conditions. Whenever an ON-unit is entered, considerable error-handling overhead is incurred. To implement the logic, you should perform the necessary tests, rather than use the compiler's condition-detecting facilities.

For example, in a program using record-oriented output to a keyed data set, you might wish to eliminate certain keys because they would not fit into the limits of the data set. You can rely on the raising of the KEY condition to detect unsuitable keys, but it is considerably more efficient to test each key yourself.

- The SIZE condition might not be raised if a floating-point variable with a value between 2,147,483,648 and 4,294,967,295 is assigned to a fixed-decimal variable with less than 10 digits.

---

## Notes about multitasking

The use of bit strings in a multitasking program can occasionally cause incorrect results. When the program references the bit string, it might be necessary for a PL/I library routine to access adjacent storage, as well as the string itself. If another task modifies this adjacent storage at the same time, the results can be unpredictable. The problem is less likely to occur with aligned bit strings than unaligned.



---

## Part 5. Using interfaces to other products

<b>Chapter 15. Using the Sort Program</b> . . . . .	348
Preparing to Use Sort . . . . .	348
Choosing the Type of Sort . . . . .	349
Specifying the Sorting Field . . . . .	352
Specifying the Records to be Sorted . . . . .	354
Maximum Record Lengths . . . . .	354
Determining Storage Needed for Sort . . . . .	355
Main Storage . . . . .	355
Auxiliary Storage . . . . .	355
Calling the Sort Program . . . . .	355
Example 1 . . . . .	357
Example 2 . . . . .	357
Example 3 . . . . .	357
Example 4 . . . . .	357
Example 5 . . . . .	357
Determining Whether the Sort Was Successful . . . . .	358
Establishing Data Sets for Sort . . . . .	358
Sort Work Data Sets . . . . .	359
Input Data Set . . . . .	359
Output Data Set . . . . .	359
Checkpoint Data Set . . . . .	360
Sort Data Input and Output . . . . .	360
Data Input and Output Handling Routines . . . . .	360
E15 — Input Handling Routine (Sort Exit E15) . . . . .	361
E35 — Output Handling Routine (Sort Exit E35) . . . . .	364
Calling PLISRTA Example . . . . .	365
Calling PLISRTB Example . . . . .	366
Calling PLISRTC Example . . . . .	367
Calling PLISRTD Example . . . . .	368
Sorting Variable-Length Records Example . . . . .	369

---

## Chapter 15. Using the Sort Program

The compiler provides an interface called PLISRT $x$  ( $x = A, B, C,$  or  $D$ ) that allows you to make use of the IBM-supplied Sort programs in MVS and VM.

To use the MVS or VM Sort program with PLISRT $x$ , you must:

1. Include a call to one of the entry points of PLISRT $x$ , passing it the information on the fields to be sorted. This information includes the length of the records, the maximum amount of storage to use, the name of a variable to be used as a return code, and other information required to carry out the sort.
2. Specify the data sets required by the Sort program in JCL DD statements or by use of the ALLOCATE command on TSO and FILEDEF commands on VM.

When used from PL/I, the Sort program sorts records of all normal lengths on a large number of sorting fields. Data of most types can be sorted into ascending or descending order. The source of the data to be sorted can be either a data set or a user-written PL/I procedure that the Sort program will call each time a record is required for the sort. Similarly, the destination of the sort can be a data set or a PL/I procedure that handles the sorted records.

Using PL/I procedures allows processing to be done before or after the sort itself, thus allowing a complete sorting operation to be handled completely by a call to the sort interface. It is important to understand that the PL/I procedures handling input or output are called from the Sort program itself and will effectively become part of it.

PL/I can operate with DFSORT or a program with the same interface. DFSORT is a release of the program product 5740-SM1. PL/I can also operate with DFSORT/CMS. DFSORT has many built-in features you can use to eliminate the need for writing program logic (for example, INCLUDE, OMIT, OUTREC and SUM statement plus the many ICETOOL operators). See *DFSORT Application Programming Guide* for details and *Getting Started with DFSORT* for a tutorial.

**Note:** None of your routines should have the name SORT if you are using DFSORT/CMS.

The following material applies to DFSORT. Because you can use programs other than DFSORT, the actual capabilities and restrictions vary. For these capabilities and restrictions, see *DFSORT Application Programming Guide*, or the equivalent publication for your sort product.

To use the Sort program you must include the correct PL/I statements in your source program and specify the correct data sets in your JCL, or in your TSO ALLOCATE or VM FILEDEF commands.

---

### Preparing to Use Sort

Before using Sort, you must determine the type of sort you require, the length and format of the sorting fields in the data, the length of your data records, and the amount of auxiliary and main storage you will allow for sorting.

To determine the PLISRT<sub>x</sub> entry point that you will use, you must decide the source of your unsorted data, and the destination of your sorted data. You must choose between data sets and PL/I subroutines. Using data sets is simpler to understand and gives faster performance. Using PL/I subroutines gives you more flexibility and more function, enabling you to manipulate or print the data before it is sorted, and to make immediate use of it in its sorted form. If you decide to use an input or output handling subroutine, you will need to read “Data Input and Output Handling Routines” on page 360.

The entry points and the source and destination of data are as follows:

Entry point	Source	Destination
PLISRTA	Data set	Data set
PLISRTB	Subroutine	Data set
PLISRTC	Data set	Subroutine
PLISRTD	Subroutine	Subroutine

Having determined the entry point you are using, you must now determine the following things about your data set:

- The position of the sorting fields; these can be either the complete record or any part or parts of it
- The type of data these fields represent, for example, character or binary
- Whether you want the sort on each field to be in ascending or descending order
- Whether you want equal records to be retained in the order of the input, or whether their order can be altered during sorting

Specify these options on the SORT statement, which is the first argument to PLISRT<sub>x</sub>. After you have determined these, you must determine two things about the records to be sorted:

- Whether the record format is fixed or varying
- The length of the record, which is the maximum length for varying

Specify these on the RECORD statement, which is the second argument to PLISRT<sub>x</sub>.

Finally, you must decide on the amount of main and auxiliary storage you will allow for the Sort program. For further details, see “Determining Storage Needed for Sort” on page 355.

## Choosing the Type of Sort

If you want to make the best use of the Sort program, you must understand something of how it works. In your PL/I program you specify a sort by using a CALL statement to the sort interface subroutine PLISRT<sub>x</sub>. This subroutine has four entry points: *x*=A, B, C, and D. Each specifies a different source for the unsorted data and destination for the data when it has been sorted. For example, a call to PLISRTA specifies that the unsorted data (the input to sort) is on a data set, and that the sorted data (the output from sort) is to be placed on another data set. The CALL PLISRT<sub>x</sub> statement must contain an argument list giving the Sort program information about the data set to be sorted, the fields on which it is to be sorted,

the amount of space available, the name of a variable into which Sort will place a return code indicating the success or failure of the sort, and the name of any output or input handling procedure that can be used.

The sort interface routine builds an argument list for the Sort program from the information supplied by the PLISRTx argument list and the choice of PLISRTx entry point. Control is then transferred to the Sort program. If you have specified an output- or input-handling routine, this will be called by the Sort program as many times as is necessary to handle each of the unsorted or sorted records. When the sort operation is complete, the Sort program returns to the PL/I calling procedure communicating its success or failure in a return code, which is placed in one of the arguments passed to the interface routine. The return code can then be tested in the PL/I routine to discover whether processing should continue. Figure 82 on page 351 is a simplified flowchart showing this operation.

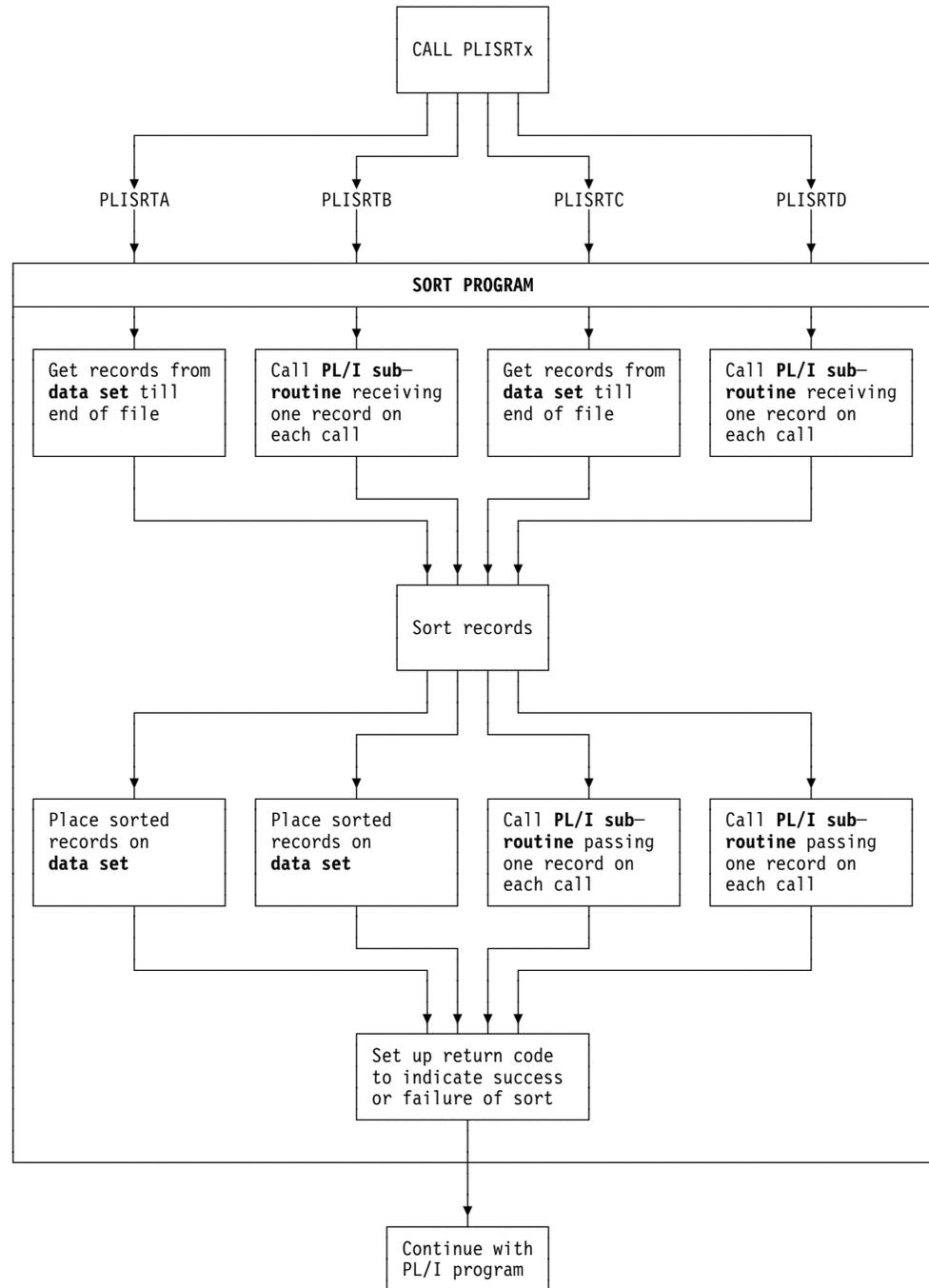


Figure 82. Flow of Control for Sort Program

Within the Sort program itself, the flow of control between the Sort program and input- and output-handling routines is controlled by return codes. The Sort program calls these routines at the appropriate point in its processing. (Within the Sort program, and its associated documentation, these routines are known as *user exits*. The routine that passes input to be sorted is the E15 sort user exit. The routine that processes sorted output is the E35 sort user exit.) From the routines, Sort expects a return code indicating either that it should call the routine again, or that it should continue with the next stage of processing.

The important points to remember about Sort are: (1) it is a self-contained program that handles the complete sort operation, and (2) it communicates with the caller, and with the user exits that it calls, by means of return codes.

The remainder of this chapter gives detailed information on how to use Sort from PL/I. First the required PL/I statements are described, and then the data set requirements. The chapter finishes with a series of examples showing the use of the four entry points of the sort interface routine.

## Specifying the Sorting Field

The SORT statement is the first argument to PLISRTx. The syntax of the SORT statement must be a character string expression that takes the form:

```
'bSORTbFIELDS=(start1,length1,form1,seq1,  
...startn,lengthn,formn,seqn)[,other options]b'
```

For example:

```
' SORT FIELDS=(1,10,CH,A) '
```

**b** represents one or more blanks. Blanks shown are mandatory. No other blanks are allowed.

### **start,length,form,seq**

defines a sorting field. You can specify any number of sorting fields, but there is a limit on the total length of the fields. If more than one field is to be sorted on, the records are sorted first according to the first field, and then those that are of equal value are sorted according to the second field, and so on. If all the sorting values are equal, the order of equal records will be arbitrary unless you use the EQUALS option. (See later in this definition list.) Fields can overlay each other.

For DFSORT (5740-SM1), the maximum total length of the sorting fields is restricted to 4092 bytes and all sorting fields must be within 4092 bytes of the start of the record. Other sort products might have different restrictions.

**start** is the starting position within the record. Give the value in bytes except for binary data where you can use a "byte.bit" notation. The first byte in a string is considered to be byte 1, the first bit is bit 0. (Thus the second bit in byte 2 is referred to as 2.1.) For varying length records you must include the 4-byte length prefix, making 5 the first byte of data.

**length** is the length of the sorting field. Give the value in bytes except for binary where you can use "byte.bit" notation. The length of sorting fields is restricted according to their data type.

**form** is the format of the data. This is the format assumed for the purpose of sorting. All data passed between PL/I routines and Sort must be in the form of character strings. The main data types and the restrictions on their length are shown below. Additional data types are available for special-purpose sorts. See the *DFSORT Application Programming Guide*, or the equivalent publication for your sort product.

**Code Data type and length**

CH	character	1–4096
ZD	zoned decimal, signed	1–32
PD	packed decimal, signed	1–32
FI	fixed point, signed	1–256
BI	binary, unsigned	1 bit to 4092 bytes
FL	floating-point, signed	1–256
FS	floating-sign,	1–16

The sum of the lengths of all fields must not exceed 4092 bytes.

**seq** is the sequence in which the data will be sorted as follows:

- A ascending (that is, 1,2,3,...)
- D descending (that is, ...,3,2,1)

**Note:** You cannot specify E, because PL/I does not provide a method of passing a user-supplied sequence.

**other options**

You can specify a number of other options, depending on your Sort program. You must separate them from the FIELDS operand and from each other by commas. Do not place blanks between operands.

**FILSZ=y**

specifies the number of records in the sort and allows for optimization by Sort. If *y* is only approximate, E should precede *y*.

**SKIPREC=y**

specifies that *y* records at the start of the input file are to be ignored before sorting the remaining records.

**CKPT or CHKPT**

specifies that checkpoints are to be taken. If you use this option, you must provide a SORTCKPT data set and DFSORT's 16NCKPT=NO installation option must be specified.

**EQUALS|NOEQUALS**

specifies whether the order of equal records will be preserved as it was in the input (EQUALS) or will be arbitrary (NOEQUALS). You could improve sort performance by using the NOEQUALS. The default option is chosen when Sort is installed. The IBM-supplied default is NOEQUALS.

**DYNALLOC=(d,n)**

(OS/VS Sort only) specifies that the program dynamically allocates intermediate storage.

- d** is the device type (3380, etc.).
- n** is the number of work areas.

## Specifying the Records to be Sorted

Use the RECORD statement as the second argument to PLISRTx. The syntax of the RECORD statement must be a character string expression which, when evaluated, takes the syntax shown below:

```
'bRECORDbTYPE=rectype[,LENGTH=(L1,[,L4,L5])]b'
```

For example:

```
' RECORD TYPE=F,LENGTH=(80) '
```

**b** represents one or more blanks. Blanks shown are mandatory. No other blanks are allowed.

### TYPE

specifies the type of record as follows:

**F** fixed length  
**V** varying length EBCDIC  
**D** varying length ASCII

Even when you use input and output routines to handle the unsorted and sorted data, you must specify the record type as it applies to the work data sets used by Sort.

If varying length strings are passed to Sort from an input routine (E15 exit), you should normally specify V as a record format. However, if you specify F, the records are padded to the maximum length with blanks.

### LENGTH

specifies the length of the record to be sorted. You can omit LENGTH if you use PLISRTA or PLISRTC, because the length will be taken from the input data set. Note that there is a restriction on the maximum and minimum length of the record that can be sorted (see below). For varying length records, you must include the four-byte prefix.

- 11** is the length of the record to be sorted. For VSAM data sets sorted as varying records it is the maximum record size+4.
- ,,** represent two arguments that are not applicable to Sort when called from PL/I. You must include the commas if the arguments that follow are used.
- 14** specifies the minimum length of record when varying length records are used. If supplied, it is used by Sort for optimization purposes.
- 15** specifies the modal (most common) length of record when varying length records are used. If supplied, it is used by Sort for optimization purposes.

### Maximum Record Lengths

The length of a record can never exceed the maximum length specified by the user. The maximum record length for variable length records is 32756 bytes, for fixed length records it is 32760 bytes, and for spanned records it is 32767 bytes.

The minimum block length for tape work units (which should be avoided for performance reasons) is 18 bytes; the minimum record length is 14 bytes.

## Determining Storage Needed for Sort

### Main Storage

Sort requires both main and auxiliary storage. The minimum main storage for DFSORT is 88K bytes, but for best performance, more storage (on the order of 1 megabyte or more) is recommended. DFSORT can take advantage of storage above 16M virtual or extended architecture processors. Under MVS/ESA, DFSORT can also take advantage of expanded storage. You can specify that Sort use the maximum amount of storage available by passing a storage parameter in the following manner:

```
DCL MAXSTOR FIXED BINARY (31,0);
UNSPEC(MAXSTOR)='00000000'B|UNSPEC('MAX');
CALL PLISRTA
  (' SORT FIELDS=(1,80,CH,A) ',
   ' RECORD TYPE=F,LENGTH=(80) ',
   MAXSTOR,
   RETCODE,
   'TASK');
```

If files are opened in E15 or E35 exit routines, enough residual storage should be allowed for the files to open successfully.

### Auxiliary Storage

Calculating the minimum auxiliary storage for a particular sorting operation is a complicated task. To achieve maximum efficiency with auxiliary storage, use direct access storage devices (DASDs) whenever possible. For more information on improving program efficiency, see the *DFSORT Application Programming Guide*, particularly the information about dynamic allocation of workspace which allows DFSORT to determine the auxiliary storage needed and allocate it for you.

If you are interested only in providing enough storage to ensure that the sort will work, make the total size of the SORTWK data sets large enough to hold three sets of the records being sorted. (You will not gain any advantage by specifying more than three if you have enough space in three data sets.)

However, because this suggestion is an approximation, it might not work, so you should check the sort manuals. If this suggestion does work, you will probably have wasted space.

---

## Calling the Sort Program

When you have determined the points described above, you are in a position to write the CALL PLISRTx statement. You should do this with some care; for the entry points and arguments to use, see Table 44.

---

Table 44 (Page 1 of 2). The Entry Points and Arguments to PLISRTx (x = A, B, C, or D)

Entry points	Arguments
PLISRTA Sort input: data set Sort output: data set	(sort statement,record statement,storage,return code [,data set prefix,message level, sort technique])
PLISRTB Sort input: PL/I subroutine Sort output: data set	(sort statement,record statement,storage,return code,input routine [,data set prefix,message level,sort technique])

Table 44 (Page 2 of 2). The Entry Points and Arguments to PLISRTx (x = A, B, C, or D)

Entry points	Arguments
PLISRTC Sort input: data set Sort output: PL/I subroutine	(sort statement,record statement,storage,return code,output routine [,data set prefix,message level,sort technique])
PLISRTD Sort input: PL/I subroutine Sort output: PL/I subroutine	(sort statement,record statement,storage,return code,input routine,output routine[,data set prefix,message level,sort technique])
Sort statement	Character string expression containing the Sort program SORT statement. Describes sorting fields and format. See "Specifying the Sorting Field" on page 352.
Record statement	Character string expression containing the Sort program RECORD statement. Describes the length and record format of data. See "Specifying the Records to be Sorted" on page 354.
Storage	Fixed binary expression giving maximum amount of main storage to be used by the Sort program. Must be >88K bytes for DFSORT. See also "Determining Storage Needed for Sort."
Return code	Fixed binary variable of precision (31,0) in which Sort places a return code when it has completed. The meaning of the return code is:  0=Sort successful 16=Sort failed 20=Sort message data set missing
Input routine	(PLISRTB and PLISRTD only.) Name of the PL/I external or internal procedure used to supply the records for the Sort program at sort exit E15.
Output routine	(PLISRTC and PLISRTD only.) Name of the PL/I external or internal procedure to which Sort passes the sorted records at sort exit E35.
Data set prefix	Character string expression of four characters that replaces the default prefix of 'SORT' in the names of the sort data sets SORTIN, SORTOUT, SORTWKnn and SORTCNTL, if used. Thus if the argument is "TASK", the data sets TASKIN, TASKOUT, TASKWKnn, and TASKCNTL can be used. This facility enables multiple invocations of Sort to be made in the same job step. The four characters must start with an alphabetic character and must not be one of the reserved names PEER, BALN, CRCX, OSCL, POLY, DIAG, SYSC, or LIST. You must code a null string for this argument if you require either of the following arguments but do not require this argument.
Message level	Character string expression of two characters indicating how Sort's diagnostic messages are to be handled, as follows:  NO No messages to SYSOUT AP All messages to SYSOUT CP Critical messages to SYSOUT  SYSOUT will normally be allocated to the printer, hence the use of the mnemonic letter "P". Other codes are also allowed for certain of the Sort programs. For further details on these codes, see <i>DFSORT Application Programming Guide</i> . You must code a null string for this argument if you require the following argument but you do not require this argument.
Sort technique	(This is not used by DFSORT; it appears for compatibility reasons only.) Character string of length 4 that indicates the type of sort to be carried out, as follows:  PEER Peerage sort BALN Balanced CRCX Criss-cross sort OSCL Oscillating POLY Polyphase sort  Normally the Sort program will analyze the amount of space available and choose the most effective technique without any action from you. You should use this argument only as a bypass for sorting problems or when you are certain that performance could be improved by another technique. See <i>DFSORT Application Programming Guide</i> for further information.

The examples below indicate the form that the CALL PLISRTx statement normally takes.

### Example 1

A call to PLISRTA sorting 80-byte records from SORTIN to SORTOUT using 1048576 (1 megabyte) of storage, and a return code, RETCODE, declared as FIXED BINARY (31,0).

```
CALL PLISRTA (' SORT FIELDS=(1,80,CH,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
              1048576,
              RETCODE);
```

### Example 2

This example is the same as example 1 except that the input, output, and work data sets are called TASKIN, TASKOUT, TASKWK01, and so forth. This might occur if Sort was being called twice in one job step.

```
CALL PLISRTA (' SORT FIELDS=(1,80,CH,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
              1048576,
              RETCODE,
              'TASK');
```

### Example 3

This example is the same as example 1 except that the sort is to be undertaken on two fields. First, bytes 1 to 10 which are characters, and then, if these are equal, bytes 11 and 12 which contain a binary field, both fields are to be sorted in ascending order.

```
CALL PLISRTA (' SORT FIELDS=(1,10,CH,A,11,2,BI,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
              1048576,
              RETCODE);
```

### Example 4

This example shows a call to PLISRTB. The input is to be passed to Sort by the PL/I routine PUTIN, the sort is to be carried out on characters 1 to 10 of an 80 byte fixed length record. Other information as above.

```
CALL PLISRTB (' SORT FIELDS=(1,10,CH,A) ',
              ' RECORD TYPE=F,LENGTH=(80) ',
              1048576,
              RETCODE,
              PUTIN);
```

### Example 5

This example shows a call to PLISRTD. The input is to be supplied by the PL/I routine PUTIN and the output is to be passed to the PL/I routine PUTOUT. The record to be sorted is 84 bytes varying (including the length prefix). It is to be sorted on bytes 1 through 5 of the data in ascending order, then if these fields are equal, on bytes 6 through 10 in descending order. (Note that the 4-byte length prefix is included so that the actual values used are 5 and 10 for the starting points.) If both these fields are the same, the order of the input is to be retained. (The EQUALS option does this.)

```

CALL PLISRTD (' SORT FIELDS=(5,5,CH,A,10,5,CH,D),EQUALS ',
             ' RECORD TYPE=V,LENGTH=(84) ',
             1048576,
             RETCODE,
             PUTIN,      /*input routine (sort exit E15)*/
             PUTOUT);   /*output routine (sort exit E35)*/

```

## Determining Whether the Sort Was Successful

When the sort is completed, Sort sets a return code in the variable named in the fourth argument of the call to PLISRTx. It then returns control to the statement that follows the CALL PLISRTx statement. The value returned indicates the success or failure of the sort as follows:

0	Sort successful
16	Sort failed
20	Sort message data set missing

You must declare the variable to which the return code is passed as FIXED BINARY (31,0). It is standard practice to test the value of the return code after the CALL PLISRTx statement and take appropriate action according to the success or failure of the operation.

For example (assuming the return code was called RETCODE):

```

IF RETCODE≠0 THEN DO;
  PUT DATA(RETCODE);
  SIGNAL ERROR;
END;

```

If the job step that follows the sort depends on the success or failure of the sort, you should set the value returned in the Sort program as the return code from the PL/I program. This return code is then available for the following job step. The PL/I return code is set by a call to PLIRETC. You can call PLIRETC with the value returned from Sort thus:

```

CALL PLIRETC(RETCODE);

```

You should not confuse this call to PLIRETC with the calls made in the input and output routines, where a return code is used for passing control information to Sort.

## Establishing Data Sets for Sort

If DFSORT was installed in a library not known to the system, you must specify the DFSORT library in a JOBLIB or STEPLIB DD statement.

When you call Sort, certain sort data sets must not be open. These are:

### **SORTLIB**

This library is only required if your work data sets (see below) are on magnetic tape (which is not recommended for performance reasons). You must get the name of this data set from your system programmer.

### **SYSOUT**

A data set (normally the printer) on which messages from the Sort program will be written.

## **Sort Work Data Sets**

### **SORTWK01–SORTWK32**

**Note:** If you specify more than 32 sort work data sets, DFSORT will only use the first 32.

### **\*\*\*\*WK01–\*\*\*\*WK32**

From 1 to 32 working data sets used in the sorting process. These can be direct-access or on magnetic tape. For a discussion of space required and number of data sets, see “Determining Storage Needed for Sort” on page 355.

\*\*\*\* represents the four characters that you can specify as the data set prefix argument in calls to PLISRTx. This allows you to use data sets with prefixes other than SORT. They must start with an alphabetic character and must not be the names PEER, BALN, CRCX, OSCL, POLY, SYSC, LIST, or DIAG.

## **Input Data Set**

### **SORTIN**

#### **\*\*\*\*IN**

The input data set used when PLISRTA and PLISRTC are called.

See \*\*\*\*WK01–\*\*\*\*WK32 above for a detailed description.

## **Output Data Set**

### **SORTOUT**

#### **\*\*\*\*OUT**

The output data set used when PLISRTA and PLISRTB are called.

See \*\*\*\*WK01–\*\*\*\*WK32 above for a detailed description.

## Checkpoint Data Set

### **SORTCKPT**

Data set used to hold checkpoint data, if CKPT or CHKPT option was used in the SORT statement argument and DFSORT's 16NCKPT=NO installation option was specified. For information on this program DD statement, see *DFSORT Application Programming Guide*.

### **DFSPARM SORTCNTL**

Data set from which additional or changed control statements can be read (optional). For additional information on this program DD statement, see *DFSORT Application Programming Guide*.

See \*\*\*\*WK01–\*\*\*\*WK32 above for a detailed description.

---

## Sort Data Input and Output

The source of the data to be sorted is provided either directly from a data set or indirectly by a routine (Sort Exit E15) written by the user. Similarly, the destination of the sorted output is either a data set or a routine (Sort Exit E35) provided by the user.

PLISRTA is the simplest of all of the interfaces because it sorts from data set to data set. An example of a PLISRTA program is in Figure 86 on page 365. Other interfaces require either the input handling routine or the output handling routine, or both.

---

## Data Input and Output Handling Routines

The input handling and output handling routines are called by Sort when PLISRTB, PLISRTC, or PLISRTD is used. They must be written in PL/I, and can be either internal or external procedures. If they are internal to the routine that calls PLISRTx, they behave in the same way as ordinary internal procedures in respect of scope of names. The input and output procedure names must themselves be known in the procedure that makes the call to PLISRTx.

The routines are called individually for each record required by Sort or passed from Sort. Therefore, each routine must be written to handle one record at a time. Variables declared as AUTOMATIC within the procedures will not retain their values between calls. Consequently, items such as counters, which need to be retained from one call to the next, should either be declared as STATIC or be declared in the containing block.

## E15 — Input Handling Routine (Sort Exit E15)

Input routines are normally used to process the data in some way before it is sorted. You can use input routines to print the data, as shown in the Figure 87 on page 366 and Figure 89 on page 368, or to generate or manipulate the sorting fields to achieve the correct results.

The input handling routine is used by Sort when a call is made to either PLISRTB or PLISRTD. When Sort requires a record, it calls the input routine which should return a record in character string format, with a return code of 12. This return code means that the record passed is to be included in the sort. Sort continues to call the routine until a return code of 8 is passed. A return code of 8 means that all records have *already* been passed, and that Sort is not to call the routine again. If a record is returned when the return code is 8, it is ignored by Sort.

The data returned by the routine must be a character string. It can be fixed or varying. If it is varying, you should normally specify V as the record format in the RECORD statement which is the second argument in the call to PLISRTx. However, you can specify F, in which case the string will be padded to its maximum length with blanks. The record is returned with a RETURN statement, and you must specify the RETURNS attribute in the PROCEDURE statement. The return code is set in a call to PLIRETC. A flowchart for a typical input routine is shown in Figure 83 on page 362.

Skeletal code for a typical input routine is shown in Figure 84 on page 363.

Input Handling Subroutine

Output Handling Subroutine

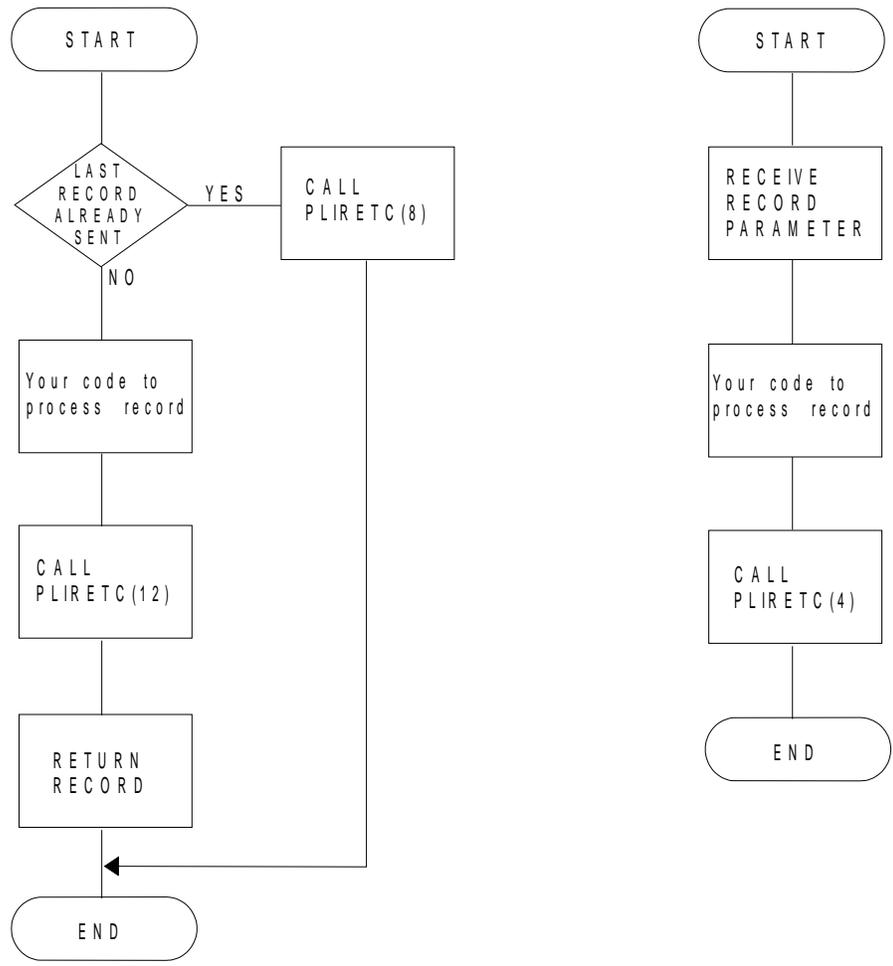


Figure 83. Flowcharts for Input and Output Handling Subroutines

---

```

E15: PROC RETURNS (CHAR(80));
      /*-----*/
      /*RETURNS attribute must be used specifying length of data to be */
      /* sorted, maximum length if varying strings are passed to Sort. */
      /*-----*/
DCL STRING CHAR(80); /*-----*/
                      /*A character string variable will normally be*/
                      /* required to return the data to Sort      */
                      /*-----*/

IF LAST_RECORD_SENT THEN
DO;
  /*-----*/
  /*A test must be made to see if all the records have been sent, */
  /*if they have, a return code of 8 is set up and control returned*/
  /*to Sort                                                         */
  /*-----*/

      CALL PLIRETC(8); /*-----*/
                      /* Set return code of 8, meaning last record */
                      /* already sent.                               */
                      /*-----*/

      GOTO FINAL;

END;

ELSE
DO;
  /*-----*/
  /* If another record is to be sent to Sort, do the*/
  /* necessary processing, set a return code of 12 */
  /* by calling PLIRETC, and return the data as a */
  /* character string to Sort                       */
  /*-----*/

  ****(The code to do your processing goes here)

      CALL PLIRETC (12); /*-----*/
                      /* Set return code of 12, meaning this */
                      /* record is to be included in the sort */
                      /*-----*/

      RETURN (STRING); /*Return data with RETURN statement*/
END;
FINAL:
END; /*End of the input procedure*/

```

---

Figure 84. Skeletal Code for an Input Procedure

Examples of an input routine are given in Figure 87 on page 366 and Figure 89 on page 368.

In addition to the return codes of 12 (include current record in sort) and 8 (all records sent), Sort allows the use of a return code of 16. This ends the sort and causes Sort to return to your PL/I program with a return code of 16—Sort failed.

**Note:** A call to PLIRETC sets a return code that will be passed by your PL/I program, and will be available to any job steps that follow it. When an output handling routine has been used, it is good practice to reset the return code with a call to PLIRETC after the call to PLISRTx to avoid receiving a nonzero completion code. By calling PLIRETC with the return code from Sort as the argument, you can make the PL/I return code reflect the success or failure of the sort. This practice is shown in Figure 88 on page 367.

## E35 — Output Handling Routine (Sort Exit E35)

Output handling routines are normally used for any processing that is necessary after the sort. This could be to print the sorted data, as shown in Figure 88 on page 367 and Figure 89 on page 368, or to use the sorted data to generate further information. The output handling routine is used by Sort when a call is made to PLISRTC or PLISRTD. When the records have been sorted, Sort passes them, one at a time, to the output handling routine. The output routine then processes them as required. When all the records have been passed, Sort sets up its return code and returns to the statement after the CALL PLISRTx statement. There is no indication from Sort to the output handling routine that the last record has been reached. Any end-of-data handling must therefore be done in the procedure that calls PLISRTx.

The record is passed from Sort to the output routine as a character string, and you must declare a character string parameter in the output handling subroutine to receive the data. The output handling subroutine must also pass a return code of 4 to Sort to indicate that it is ready for another record. You set the return code by a call to PLIRETC.

The sort can be stopped by passing a return code of 16 to Sort. This will result in Sort returning to the calling program with a return code of 16—Sort failed.

The record passed to the routine by Sort is a character string parameter. If you specified the record type as F in the second argument in the call to PLISRTx, you should declare the parameter with the length of the record. If you specified the record type as V, you should declare the parameter as adjustable, as in the following example:

```
DCL STRING CHAR(*);
```

Figure 90 on page 369 shows a program that sorts varying length records.

A flowchart for a typical output handling routine is given in Figure 83 on page 362. Skeletal code for a typical output handling routine is shown in Figure 85.

---

```
E35: PROC(STRING);      /*The procedure must have a character string
                        parameter to receive the record from Sort*/

    DCL STRING CHAR(80); /*Declaration of parameter*/

    (Your code goes here)

    CALL PLIRETC(4);    /*Pass return code to Sort indicating that the next
                        sorted record is to be passed to this procedure.*/
    END E35;           /*End of procedure returns control to Sort*/
```

---

*Figure 85. Skeletal Code for an Output Handling Procedure*

You should note that a call to PLIRETC sets a return code that will be passed by your PL/I program, and will be available to any job steps that follow it. When you have used an output handling routine, it is good practice to reset the return code with a call to PLIRETC after the call to PLISRTx to avoid receiving a nonzero completion code. By calling PLIRETC with the return code from Sort as the argument, you can make the PL/I return code reflect the success or failure of the sort. This practice is shown in the examples at the end of this chapter.

## Calling PLISRTA Example

After each time that the PL/I input- and output-handling routines communicate the return-code information to the Sort program, the return-code field is reset to zero; therefore, it is not used as a regular return code other than its specific use for the Sort program.

For details on handling conditions, especially those that occur during the input- and output-handling routines, see *Language Environment for MVS & VM Programming Guide*.

---

```
//OPT14#7 JOB ...
//STEP1 EXEC IEL1CLG
//PLI.SYSIN DD *
EX106: PROC OPTIONS(MAIN);
    DCL RETURN_CODE FIXED BIN(31,0);

    CALL PLISRTA (' SORT FIELDS=(7,74,CH,A) ',
                ' RECORD TYPE=F,LENGTH=(80) ',
                1048576
                RETURN_CODE);
    SELECT (RETURN_CODE);
    WHEN(0) PUT SKIP EDIT
        ('SORT COMPLETE RETURN_CODE 0') (A);
    WHEN(16) PUT SKIP EDIT
        ('SORT FAILED, RETURN_CODE 16') (A);
    WHEN(20) PUT SKIP EDIT
        ('SORT MESSAGE DATASET MISSING ') (A);
    OTHER PUT SKIP EDIT (
        'INVALID SORT RETURN_CODE = ', RETURN_CODE) (A,F(2));
    END /* select */;
    CALL PLIRETC(RETURN_CODE);
    /*set PL/I return code to reflect success of sort*/
    END EX106;
//GO.SORTIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTOUT DD SYSOUT=A
//GO.SYSOUT DD SYSOUT=A
//GO.SORTWK01 DD UNIT=SYSDA,SPACE=(CYL,2)
/*
```

---

Figure 86. PLISRTA—Sorting from Input Data Set to Output Data Set

## Calling PLISRTB Example

```
//OPT14#8 JOB ...
//STEP1 EXEC IEL1CLG
//PLI.SYSIN DD *
EX107: PROC OPTIONS(MAIN);

    DCL RETURN_CODE FIXED BIN(31,0);

    CALL PLISRTB (' SORT FIELDS=(7,74,CH,A) ',
                ' RECORD TYPE=F,LENGTH=(80) ',
                1048576
                RETURN_CODE,
                E15X);
    SELECT(RETURN_CODE);
    WHEN(0) PUT SKIP EDIT
        ('SORT COMPLETE RETURN_CODE 0') (A);
    WHEN(16) PUT SKIP EDIT
        ('SORT FAILED, RETURN_CODE 16') (A);
    WHEN(20) PUT SKIP EDIT
        ('SORT MESSAGE DATASET MISSING ') (A);
    OTHER PUT SKIP EDIT
        ('INVALID RETURN_CODE = ',RETURN_CODE)(A,F(2));
    END /* select */;
    CALL PLIRETC(RETURN_CODE);
    /*set PL/I return code to reflect success of sort*/

E15X: /* INPUT HANDLING ROUTINE GETS RECORDS FROM THE INPUT
      STREAM AND PUTS THEM BEFORE THEY ARE SORTED*/
    PROC RETURNS (CHAR(80));
    DCL SYSIN FILE RECORD INPUT,
        INFIELD CHAR(80);

    ON ENDFILE(SYSIN) BEGIN;
    PUT SKIP(3) EDIT ('END OF SORT PROGRAM INPUT')(A);
    CALL PLIRETC(8); /* signal that last record has
                    already been sent to sort*/

    GOTO ENDE15;
    END;

    READ FILE (SYSIN) INTO (INFIELD);
    PUT SKIP EDIT (INFIELD)(A(80)); /*PRINT INPUT*/
    CALL PLIRETC(12); /* request sort to include current
                    record and return for more*/

    RETURN(INFIELD);
    ENDE15:
    END E15X;
END EX107;

/*
//GO.SYSIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTOUT DD SYSOUT=A
//GO.SYSOUT DD SYSOUT=A
/*
//GO.SORTCNTL DD *
    OPTION DYNALLOC=(3380,2),SKIPREC=2
/*
```

Figure 87. PLISRTB—Sorting from Input Handling Routine to Output Data Set

## Calling PLISRTC Example

---

```
//OPT14#9 JOB ...
//STEP1 EXEC IELICLG
//PLI.SYSIN DD *
EX108: PROC OPTIONS(MAIN);

    DCL RETURN_CODE FIXED BIN(31,0);

    CALL PLISRTC (' SORT FIELDS=(7,74,CH,A) ',
                ' RECORD TYPE=F,LENGTH=(80) ',
                1048576
                RETURN_CODE,
                E35X);
    SELECT(RETURN_CODE);
    WHEN(0) PUT SKIP EDIT
            ('SORT COMPLETE RETURN_CODE 0') (A);
    WHEN(16) PUT SKIP EDIT
            ('SORT FAILED, RETURN_CODE 16') (A);
    WHEN(20) PUT SKIP EDIT
            ('SORT MESSAGE DATASET MISSING ') (A);
    OTHER PUT SKIP EDIT
            ('INVALID RETURN_CODE = ', RETURN_CODE) (A,F(2));
    END /* select */;
    CALL PLIRETC (RETURN_CODE);
    /*set PL/I return code to reflect success of sort*/

E35X: /* output handling routine prints sorted records*/
PROC (INREC);
    DCL INREC CHAR(80);
    PUT SKIP EDIT (INREC) (A);
    CALL PLIRETC(4); /*request next record from sort*/
END E35X;
END EX108;

/*
//GO.STEPLIB DD DSN=SYS1.SORTLINK,DISP=SHR
//GO.SYSPRINT DD SYSOUT=A
//GO.SYSOUT DD SYSOUT=A
//GO.SORTIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SORTCNTL DD *
    OPTION DYNALLOC=(3380,2),SKIPREC=2
/*
```

---

Figure 88. PLISRTC—Sorting from Input Data Set to Output Handling Routine

## Calling PLISRTD Example

---

```
//OPT14#10 JOB ...
//STEP1 EXEC IELICLG
//PLI.SYSIN DD *
EX109: PROC OPTIONS(MAIN);
  DCL RETURN_CODE FIXED BIN(31,0);
  CALL PLISRTD (' SORT FIELDS=(7,74,CH,A) ',
               ' RECORD TYPE=F,LENGTH=(80) ',
               1048576
               RETURN_CODE,
               E15X,
               E35X);

  SELECT(RETURN_CODE);
  WHEN(0) PUT SKIP EDIT
    ('SORT COMPLETE RETURN_CODE 0') (A);
  WHEN(20) PUT SKIP EDIT
    ('SORT MESSAGE DATASET MISSING ') (A);
  OTHER PUT SKIP EDIT
    ('INVALID RETURN_CODE = ', RETURN_CODE) (A,F(2));
  END /* select */;

  CALL PLIRETC(RETURN_CODE);
  /*set PL/I return code to reflect success of sort*/

E15X: /* Input handling routine prints input before sorting*/
  PROC RETURNS(CHAR(80));
  DCL INFIELD CHAR(80);

  ON ENDFILE(SYSIN) BEGIN;
    PUT SKIP(3) EDIT ('END OF SORT PROGRAM INPUT. ',
                    'SORTED OUTPUT SHOULD FOLLOW')(A);
    CALL PLIRETC(8); /* Signal end of input to sort*/
    GOTO ENDE15;
  END;

  GET FILE (SYSIN) EDIT (INFIELD) (A(80));
  PUT SKIP EDIT (INFIELD)(A);
  CALL PLIRETC(12); /*Input to sort continues*/
  RETURN(INFIELD);
ENDE15:
  END E15X;

E35X: /* Output handling routine prints the sorted records*/
  PROC (INREC);

  DCL INREC CHAR(80);
  PUT SKIP EDIT (INREC) (A);
  NEXT: CALL PLIRETC(4); /* Request next record from sort*/
  END E35X;
END EX109;
/*
//GO.SYSOUT DD SYSOUT=A
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTWK01 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SORTWK02 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SORTWK03 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SYSIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
*/
```

---

Figure 89. PLISRTD—Sorting from Input Handling Routine to Output Handling Routine

## Sorting Variable-Length Records Example

---

```
//OPT14#11 JOB ...
//STEP1 EXEC IELICLG
//PLI.SYSIN DD *
/* PL/I EXAMPLE USING PLISRTD TO SORT VARIABLE-LENGTH
  RECORDS */

EX1306: PROC OPTIONS(MAIN);
  DCL RETURN_CODE FIXED BIN(31,0);
  CALL PLISRTD (' SORT FIELDS=(11,14,CH,A) ',
    ' RECORD TYPE=V,LENGTH=(84,,,24,44) ',
    /*NOTE THAT LENGTH IS MAX AND INCLUDES
      4 BYTE LENGTH PREFIX*/
    1048576
    RETURN_CODE,
    PUTIN,
    PUTOUT);

  SELECT(RETURN_CODE);
  WHEN(0) PUT SKIP EDIT (
    'SORT COMPLETE RETURN_CODE 0') (A);
  WHEN(16) PUT SKIP EDIT (
    'SORT FAILED, RETURN_CODE 16') (A);
  WHEN(20) PUT SKIP EDIT (
    'SORT MESSAGE DATASET MISSING ') (A);
  OTHER PUT SKIP EDIT (
    'INVALID RETURN_CODE = ', RETURN_CODE)
    (A,F(2));
  END /* SELECT */;

  CALL PLIRETC(RETURN_CODE);
  /*SET PL/I RETURN CODE TO REFLECT SUCCESS OF SORT*/
  PUTIN: PROC RETURNS (CHAR(80) VARYING);
  /*OUTPUT HANDLING ROUTINE*/
  /*NOTE THAT VARYING MUST BE USED ON RETURNS ATTRIBUTE
    WHEN USING VARYING LENGTH RECORDS*/
  DCL STRING CHAR(80) VAR;

  ON ENDFILE(SYSIN) BEGIN;
    PUT SKIP EDIT ('END OF INPUT')(A);
    CALL PLIRETC(8);
    GOTO ENDPUT;
  END;

  GET EDIT(STRING)(A(80));
  I=INDEX(STRING||' ',' ')-1; /*RESET LENGTH OF THE*/
  STRING = SUBSTR(STRING,1,I); /* STRING FROM 80 TO */
                                /* LENGTH OF TEXT IN */
                                /* EACH INPUT RECORD.*/
```

---

Figure 90 (Part 1 of 2). Sorting Varying-Length Records Using Input and Output Handling Routines

---

```

        PUT SKIP EDIT(1,STRING) (F(2),X(3),A);
        CALL PLIRETC(12);
        RETURN(STRING);
ENDPUT: END;
PUTOUT:PROC(STRING);
/*OUTPUT HANDLING ROUTINE OUTPUT SORTED RECORDS*/
DCL STRING CHAR (*);
/*NOTE THAT FOR VARYING RECORDS THE STRING
PARAMETER FOR THE OUTPUT-HANDLING ROUTINE
SHOULD BE DECLARED ADJUSTABLE BUT CANNOT BE
DECLARED VARYING*/
PUT SKIP EDIT(STRING)(A); /*PRINT THE SORTED DATA*/
CALL PLIRETC(4);
END; /*ENDS PUTOUT*/
END;

/*
//GO.SYSIN DD *
003329HOOKER S.W. RIVERDALE, SATCHWELL LANE, BACONSFIELD
002886BOOKER R.R. ROTORUA, LINKEDGE LANE, TOBLEY
003077ROOKER & SON, LITTLETON NURSERIES, SHOLTSPAR
059334HOOK E.H. 109 ELMTREE ROAD, GANNET PARK, NORTHAMPTON
073872HOME TAVERN, WESTLEIGH
000931FOREST, IVER, BUCKS
/*
//GO.SYSPRINT DD SYSOUT=A
//GO.SORTOUT DD SYSOUT=A
//GO.SYSOUT DD SYSOUT=A
//GO.SORTWK01 DD UNIT=SYSDA,SPACE=(CYL,1)
//GO.SORTWK02 DD UNIT=SYSDA,SPACE=(CYL,1)
//*/

```

---

*Figure 90 (Part 2 of 2). Sorting Varying-Length Records Using Input and Output Handling Routines*

---

## Part 6. Specialized programming tasks

<b>Chapter 16. Parameter Passing and Data Descriptors</b> .....	373
PL/I Parameter Passing Conventions .....	373
Passing Assembler Parameters .....	374
Passing MAIN Procedure Parameters .....	376
Options BYVALUE .....	378
Descriptors and Locators .....	380
Aggregate Locator .....	381
Area Locator/Descriptor .....	381
Array Descriptor .....	382
String Locator/Descriptor .....	383
Structure Descriptor .....	384
Arrays of Structures and Structures of Arrays .....	385
<b>Chapter 17. Using PLIDUMP</b> .....	386
PLIDUMP Usage Notes .....	387
<b>Chapter 18. Retaining the Run-Time Environment for Multiple Invocations</b> .....	389
Preinitializable Programs .....	389
The Interface for Preinitializable Programs .....	390
Using the Extended Parameter List (EPLIST) .....	390
Preinitializing a PL/I Program .....	393
Invoking an Alternative MAIN Routine .....	398
Using the Service Vector and Associated Routines .....	402
Using the Service Vector .....	402
Load Service Routine .....	403
Delete Service Routine .....	406
Get-Storage Service Routine .....	408
Free-Storage Service Routine .....	409
Exception Router Service Routine .....	409
Attention Router Service Routine .....	412
Message Router Service Routine .....	416
User Exits in Preinitializable Programs .....	419
The SYSTEM Option in Preinitializable Programs .....	419
Calling a Preinitializable Program under VM .....	419
Calling a Preinitializable Program under MVS .....	419
Establishing an Language Environment for MVS & VM-Enabled Assembler Routine as the MAIN Procedure .....	421
Retaining the Run-Time Environment Using Language Environment for MVS & VM-Enabled Assembler as MAIN .....	421
<b>Chapter 19. Multitasking in PL/I</b> .....	422
PL/I Multitasking Facilities .....	422
Creating PL/I Tasks .....	423
The TASK Option of the CALL Statement .....	423
Example .....	423
The EVENT Option of the CALL Statement .....	423
Examples .....	423
The PRIORITY Option of the CALL Statement .....	424
Examples .....	424

Synchronization and Coordination of Tasks . . . . .	424
Sharing Data between Tasks . . . . .	425
Sharing Files between Tasks . . . . .	425
Producing More Reliable Tasking Programs . . . . .	426
Terminating PL/I Tasks . . . . .	426
Dispatching Priority of Tasks . . . . .	427
Running Tasking Programs . . . . .	428
Sample Program 1: Multiple Independent Processes . . . . .	428
Multiple Independent Processes: Nontasking Version . . . . .	429
Multiple Independent Processes: Tasking Version . . . . .	430
Sample Program 2: Multiple Independent Computations . . . . .	432
Multiple Independent Computations: Nontasking Version . . . . .	433
Multiple Independent Computations: Tasking Version . . . . .	434
<b>Chapter 20. Interrupts and Attention Processing . . . . .</b>	<b>436</b>
Using ATTENTION ON-Units . . . . .	437
Interaction with a Debugging Tool . . . . .	437
<b>Chapter 21. Using the Checkpoint/Restart Facility . . . . .</b>	<b>438</b>
Requesting a Checkpoint Record . . . . .	439
Defining the Checkpoint Data Set . . . . .	440
Requesting a Restart . . . . .	440
Automatic Restart after a System Failure . . . . .	440
Automatic Restart within a Program . . . . .	441
Getting a Deferred Restart . . . . .	441
Modifying Checkpoint/Restart Activity . . . . .	441

---

## Chapter 16. Parameter Passing and Data Descriptors

This chapter describes PL/I parameter passing conventions and also special PL/I control blocks that are passed between PL/I routines at run time. The most important of these control blocks, called locators and descriptors, provide lengths, bounds, and sizes of certain types of argument data.

Assembler routines can communicate with PL/I routines by following the parameter passing techniques described in this chapter. This includes assembler routines that call PL/I routines and PL/I routines that call assembler routines.

For additional information about LE/370 run-time environment considerations, other than parameter passing conventions, see the *Language Environment for MVS & VM Programming Guide*. This includes run-time environment conventions and assembler macros supporting these conventions,

---

### PL/I Parameter Passing Conventions

PL/I passes arguments using two methods:

- By passing the address of the arguments in the argument list
- By imbedding the arguments in the argument list

This section discusses the first method. For information on the second method, see “Options BYVALUE” on page 378.

When arguments are passed by address between PL/I routines, register 1 points to a list of addresses that is called an argument list. Each address in the argument list occupies a fullword in storage. The last fullword in the list must have its high-order bit turned on for the last argument address to be recognized. If a function reference is used, the address of the returned value or its control block is passed as an implicit last argument. In this situation, the implicit last argument is marked as the last argument, using the high-order bit flagging.

If no arguments are passed in a CALL statement, register 1 is set to zero.

When arguments are passed between PL/I routines, what is passed varies depending upon the type of data passed. The argument list addresses are the addresses of the data items for scalar arithmetic items. For other items, where the receiving routines might expect information about length or format in addition to the data itself, locators and descriptors are used. For program control information such as files or entries, other control blocks are used.

Table 45 on page 374 shows the argument address that is passed between PL/I routines for different data types. The table also includes the effect of the ASSEMBLER option. This option is recommended. See “Passing Assembler Parameters” on page 374 for additional details.

Table 45. Argument List Addresses

Data type passed	Address passed
Arithmetic items	Arithmetic variable
Array or structure	Array or structure variable, if OPTIONS(ASM) Otherwise, aggregate locator <sup>1</sup>
String or area	String or area variable, if OPTIONS(ASM) <sup>2</sup> Otherwise, locator/descriptor <sup>1</sup>
File constant/variable	File variable <sup>3</sup>
Entry	Entry variable <sup>4</sup>
Label	Label variable <sup>5</sup>
Pointer	Pointer variable
Offset	Offset variable

**Notes:**

1. Locators and descriptors are described below in "Descriptors and Locators" on page 380.
2. With options ASSEMBLER: When an unaligned bit string is involved, the address passed points to the byte that contains the start of the bit string. For a VARYING length string, the address passed points to the 2-byte field specifying the current length of the string that precedes the string.
3. A file variable is a fullword holding the address of file control data.
4. An entry variable consists of two words. The first word has the address of the entry. The second word has the address of the save area for the immediately statically encompassing block or zero, if none.
5. A label variable consists of two words. The first word has the address of a label constant. The second word has the address of the save area of the block that owns the label at the time of assignment. A label constant consists of two words. The first word has the address of the label in the program. The second word has program control data.

## Passing Assembler Parameters

If you call an assembler routine from PL/I, the ASSEMBLER option is recommended in the declaration of the assembler entry name. For example,

```
DCL ASMRN ENTRY OPTIONS(ASSEMBLER);
DCL C CHAR(80);

CALL ASMRN(C);
```

When the ASSEMBLER option is specified, the addresses of the data items are passed directly. No PL/I locators or descriptors are involved. In the example above, the address in the argument list is that of the first character in the 80-byte character string.

For details about how argument lists are built, see "PL/I Parameter Passing Conventions" above.

An assembler routine whose entry point has been declared with the ASSEMBLER option can only be invoked by means of a CALL statement. You cannot use it as a function reference. You can avoid the use of function references by passing the returned value field as a formal argument to the assembler routine.

An assembler routine can pass back a return code to a PL/I routine in register 15. If you declare the assembler entry with the option RETCODE, the value passed back in register 15 is saved by the PL/I routine and is accessed when the built-in

function PLIRETV is used. If you do not declare the entry with the option RETCODE, any register 15 return code is ignored.

Figure 91 shows the coding for a PL/I routine that invokes an assembler routine with the option RETCODE.

---

```
P1: PROC;
  DCL A FIXED BIN(31) INIT(3);
  DCL C CHAR(8) INIT('ASM2 RTN');
  DCL AR(3) FIXED BIN(15);
  DCL ASM2 ENTRY EXTERNAL OPTIONS(ASM RETCODE);
  DCL PLIRETV BUILTIN;

  /* Invoke entry ASM2. */
  /* The argument list has three pointers. */
  /* The first pointer addresses a copy of variable A. */
  /* The second pointer addresses the storage for C. */
  /* The third pointer addresses the storage for AR. */
  CALL ASM2((A),C,AR);

  /* Check register 15 return code passed back from assembler */
  /* routine. */
  IF PLIRETV~=0 THEN STOP;

END P1;
```

---

*Figure 91. A PL/I Routine That Invokes an Assembler Routine with the Option RETCODE*

If an assembler routine calls a PL/I procedure, the use of locators and descriptors should be avoided. Although you cannot specify ASSEMBLER as an option in a PROCEDURE statement, locators and descriptors can be avoided if the procedures do not directly receive strings, areas, arrays, or structures as parameters. For example, you can pass a pointer to these items instead. If a length, bound, or size is needed, you can pass these as a separate parameter. If your assembler routine is invoking a PL/I MAIN procedure, see “Passing MAIN Procedure Parameters” on page 376 for additional considerations involving MAIN procedure parameters.

Figure 92 on page 376 shows a PL/I routine that is invoked by an assembler routine, which is assumed to be operating within the Language Environment environment.

```

ASMR    CSECT
        .
        .
*
*      Invoke procedure P2 passing four arguments.
*
        LA    1,ALIST          Register 1 has argument list
        L     15,P2           Set P2 entry address
        BALR  14,15           Invoke procedure P2
        .
        .
*
*      Argument list below contains the addresses of 4 arguments.
*
ALIST   DC    A(A)            1st argument address
        DC    A(P)            2nd argument address
        DC    A(Q)            3rd argument address
        DC    A(R+X'80000000') 4th argument address
*
A       DC    F'3'            Fixed bin(31) argument
P       DC    A(C)            Pointer (to C) argument
Q       DC    A(AR)           Pointer (to AR) argument
R       DC    A(ST)           Pointer (to ST) argument
*
C       DC    CL10'INVOKE P2 ' Character string
AR      DC    4D'0'           Array of 4 elements
ST      DC    F'1'            Structure
        DC    F'2'
        DC    D'0'
P2     DC    V(P2)            Procedure P2 entry address
END     ASMR

```

```

-----
/* This routine receives four parameters from ASMR.      */
/* The arithmetic item is received directly.             */
/* The character string, array and structure are         */
/* received using a pointer indirection.                 */
P2: PROC(A,P,Q,R);

DCL A FIXED BIN(31);
DCL (P,Q,R) POINTER;
DCL C CHAR(10) BASED(P);
DCL AR(4) FLOAT DEC(16) BASED(Q);
DCL 1 ST BASED(R),
    2 ST1 FIXED BIN(31),
    2 ST2 FIXED BIN(31),
    2 ST3 FLOAT DEC(16);
.
.
.
END P2;

```

Figure 92. A PL/I Routine That Is Invoked by an Assembler Routine

If you choose to code an assembler routine that passes or receives strings, areas, arrays or structures that require a locator or descriptor, see “Descriptors and Locators” on page 380 for their format and construction. Keep in mind, however, that doing so might affect the migration of these assembler routines in the future.

## Passing MAIN Procedure Parameters

The format of a PL/I MAIN procedure parameter list is controlled by the SYSTEM compiler option and the NOEXECOPS procedure option. The kind of coding needed also depends upon the type of parameter received by the MAIN procedure.

If the MAIN procedure receives no parameters or a single varying character string:

- It is recommended that the MAIN procedure be compiled with SYSTEM(MVS).

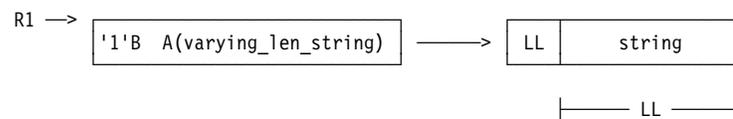
- If the assembler routine needs to pass run-time options for initializing the run-time environment, the NOEXECOPS option should not be specified or defaulted. Otherwise, NOEXECOPS should be specified.
- The MAIN procedure must be coded to have no parameters or a single parameter, consisting of a varying character string. For example:

```

MAIN: PROC(C) OPTIONS(MAIN);
      DCL C CHAR(100) VARYING;

```

- The assembler routine must invoke the MAIN procedure passing a varying length character string, as shown in Figure 93. The string has the same format as that passed in the PARM= option of the MVS EXEC statement, when an Language Environment program is executed. The string consists of optional run-time options, followed by a slash (/), followed by optional characters to be passed to the MAIN procedure.




---

Figure 93. Assembler Routine Invoking a MAIN Procedure

If NOEXECOPS is specified, run-time options and the accompanying slash should be omitted. If run-time options are provided in the string, they will have no effect on environment initialization.

If a MAIN procedure receives no parameters, the argument list should be built as shown in Figure 93 but a null string should be passed by setting the length LL to zero.

If the MAIN procedure receives a character string as a parameter, the locator/descriptor needed for this string is built by PL/I run-time services before the MAIN procedure gains control.

If a MAIN procedure receives more than one parameter or a parameter that is not a single varying character string:

- It is recommended that the MAIN procedure be compiled with SYSTEM(MVS).
- NOEXECOPS is always implied. There is no mechanism to receive and parse run-time options from the assembler routine for initializing the run-time environment.
- The assembler routine should build its argument list so it corresponds to the parameters expected by the MAIN procedure. The assembler argument list is passed through as is to the MAIN procedure. If strings, areas, arrays or structures need to be passed, consider passing pointers to these items instead. This avoids the use of locators and descriptors.

Figure 94 on page 378 illustrates this technique.

```

ASM0    CSECT
      .
      .
      .
*
*      Invoke MAINP passing two arguments.
*
      LA    1,ALIST          Register 1 has argument list
      LINK EP=MAINP         Invoke MAINP load module,
*                               giving control to entry CEESTART
      .
      .
*
*      The argument list below contains the addresses of two
*      arguments, both of which are pointers.
*
ALIST   DC    A(P)          First argument address
        DC    A(Q+X'80000000') Second argument address
*
P       DC    A(C)          Pointer (to 1st string) argument
Q       DC    A(D)          Pointer (to 2nd string) argument
*
C       DC    C'Character string 1 '
D       DC    C'Character string 2 '
        END    ASM0

```

```

-----
%PROCESS SYSTEM(MVS);

/* This procedure receives two pointers as its parameters.      */
MAINP: PROCEDURE(P,Q) OPTIONS(MAIN);
      DCL (P,Q) POINTER;
      DCL C CHAR(20) BASED(P);
      DCL D CHAR(20) BASED(Q);

      /* Display contents of character strings pointed to by    */
      /* pointer parameters.                                    */
      DISPLAY(C);
      DISPLAY(D);

      END MAINP;

```

Figure 94. Assembler Routine That Passes Pointers to Strings

- An assembler routine can choose to pass strings, areas, arrays or structures to a MAIN procedure. If so, locators and descriptors as described in “Descriptors and Locators” on page 380 must be provided by the assembler routine.

Assembler routines should not invoke MAIN procedures compiled by PL/I MVS & VM that specify SYSTEM(IMS) or SYSTEM(CICS). Assembler routines can invoke MAIN procedures compiled with other SYSTEM options. However, these interfaces generally involve specialized usage.

## Options BYVALUE

PL/I supports the BYVALUE option for external procedure entries and entry declarations. The BYVALUE option specifies that variables are passed by copying the value of the variables into the argument list. This implies that the invoked routine cannot modify the variables passed by the caller.

BYVALUE arguments and returned values must have a scalar data type of either POINTER or REAL FIXED BINARY(31,0). Consequently, each fullword slot in the argument list consists of either a pointer value or a real fixed binary(31,0) value. If

you need to pass a data type that is not POINTER or REAL FIXED BINARY(31,0), consider passing the data type indirectly using the POINTER data type.

Values from function references are returned using register 15.

With the BYVALUE argument passing convention, PL/I does not manipulate the high-order bit of BYVALUE POINTER arguments, even if last in the argument list. Further, the high-order bit is neither turned off for incoming nor set for outgoing BYVALUE arguments. Figure 95 shows an assembler routine ASM1 that invokes a procedure PX passing three arguments BYVALUE, which in turn invokes an assembler routine ASM2 (not shown). The assembler routines are assumed to be operating in the Language Environment for MVS & VM environment.

---

```

ASM1    CSECT
        .
        .
        .
*
*      Invoke procedure PX passing three arguments BYVALUE.
*
        LA    1,ALIST          Register 1 has argument list
        L     15,PX           Set reg 15 with entry point
        BALR  14,15           Invoke BYVALUE procedure
        LTR   15,15           Checked returned value
        .
        .
*
*      The BYVALUE argument list contains three arguments.
*      The high order bit of the last argument is *not* specially
*      flagged when using the BYVALUE passing convention.
*
ALIST   DC    A(A)            1st arg is pointer value
        DC    A(C)            2nd arg is pointer value
        DC    F'2'           3rd arg is arithmetic value
*
A       DC    2D'0'           Array
C       DC    C'CHARSTRING'   Character string
*
*
PX      DC    V(PX)           Entry point to be invoked
        END   ASM1

```

---

```

PX:     PROCEDURE(P,Q,M) OPTIONS(BYVALUE) RETURNS(FIXED BIN(31));
        DCL (P,Q) POINTER;
        DCL A(2) FLOAT DEC(16) BASED(P);
        DCL C   CHAR(10)   BASED(Q);
        DCL M FIXED BIN(31);
        DCL ASM2 ENTRY(FIXED BIN(31,0)) OPTIONS(BYVALUE ASM);

        M=A(1);

        /* ASM2 is passed variable M byvalue, so it can not alter */
        /* the contents of variable M.                               */
        CALL ASM2(M);

        RETURN(0);

        END PX;

```

---

Figure 95. Assembler Routine Passing Arguments BYVALUE

---

## Descriptors and Locators

PL/I supports locators and descriptors in order to communicate the lengths, bounds, and sizes of strings, areas, arrays and structures between PL/I routines. For example, the procedure below can receive the parameters shown without explicit knowledge of their lengths, bounds or sizes at compilation time.

```
P: PROCEDURE(C,D,AR);  
   DCL C CHAR(*),  
       D AREA(*),  
       AR(*,*) CHAR(*);
```

The use of locators and descriptors is not recommended for assembler routines, because the migration of these routines might be affected in the future. In addition, portability to other platforms can be adversely affected. See "Passing Assembler Parameters" on page 374 for techniques to avoid the use of these control blocks.

The major control blocks are:

**Descriptors** These hold the extent of the data item such as string lengths, array bounds, and area sizes.

**Locators** These hold the address of a data item. If they are not concatenated with the descriptor, they hold the descriptor's address.

**Locator/descriptor**

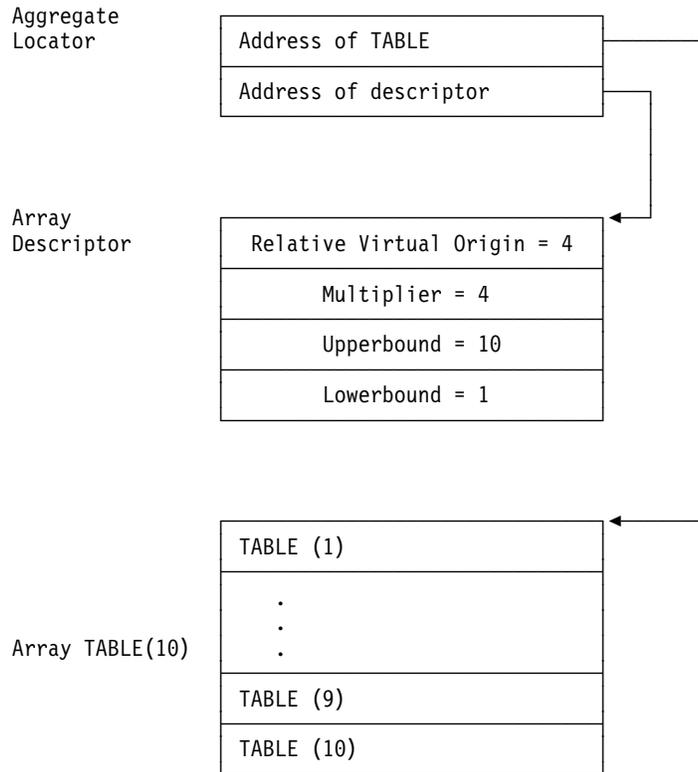
This is a control block consisting of a locator concatenated with a descriptor.

When received as parameters or passed as arguments, locators and descriptors are needed for strings, areas, arrays, and structures. For strings and areas, the locator is concatenated with the descriptor and contains only the address of the variable. For structures and arrays, the locator is a separate control block from the descriptor and holds the address of both the variable and the descriptor.

Figure 96 on page 381 gives an example of the way in which data is related to its locator and descriptor.

---

PL/I Statement: DCL TABLE(10) FLOAT DECIMAL(6);



---

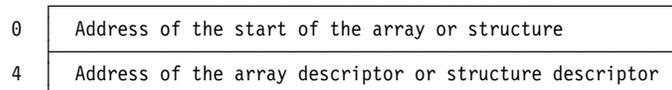
Figure 96. Example of Locator, Descriptor, and Array Storage

## Aggregate Locator

The aggregate locator holds this information:

- The address of the start of the array or structure
- The address of the array descriptor or structure descriptor

Figure 97 shows the format of the aggregate locator.



---

Figure 97. Format of the Aggregate Locator

The array and structure descriptor are described in subsequent sections.

## Area Locator/Descriptor

The area locator/descriptor holds this information:

- The address of the start of the area
- The length of the area

Figure 98 on page 382 shows the format of the area locator/descriptor.

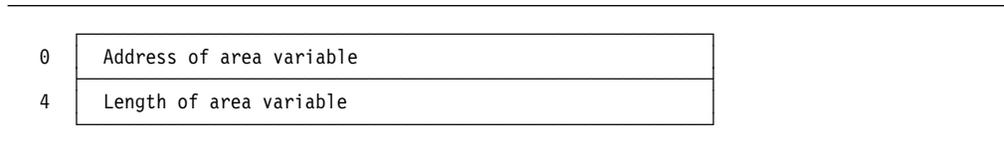


Figure 98. Format of the Area Locator/Descriptor

The area variable consists of a 16-byte area variable control block followed by the storage for the area variable.

The area descriptor is the second word of the area locator/descriptor. It is used in structure descriptors when areas appear in structures and also in array descriptors, for arrays of areas.

## Array Descriptor

The array descriptor holds this information:

- The relative virtual origin (RVO) of the array
- The multiplier for each dimension
- The high and low bounds for the subscripts in each dimension

When the array is an array of strings or areas, the string or area descriptor is concatenated at the end of the array descriptor to provide the necessary additional information. String and area descriptors are the second word of the locator/descriptor word pair.

The CMPAT(V2) format of the array descriptor differs from the CMPAT(V1) level, but holds the same information. After the first fullword (RVO) of the array descriptor, each array dimension is represented by three words (CMPAT(V2)) or two words (CMPAT(V1)) that contain the multiplier and bounds for each dimension.

Figure 99 shows the format of the CMPAT(V2) array descriptor.

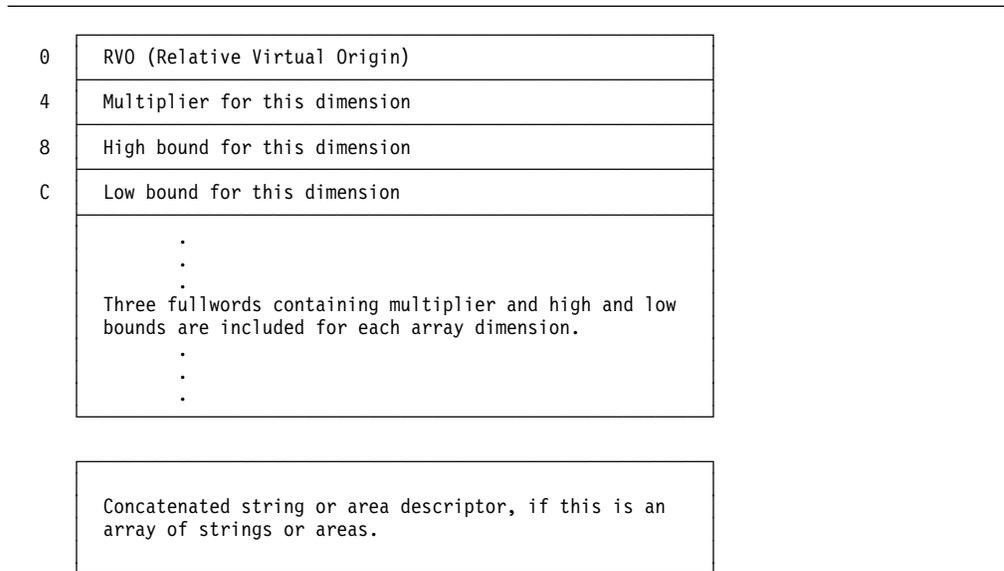


Figure 99. Format of the Array Descriptor for a Program Compiled with CMPAT(V2)

Figure 100 on page 383 shows the format of the CMPAT(V1) array descriptor.

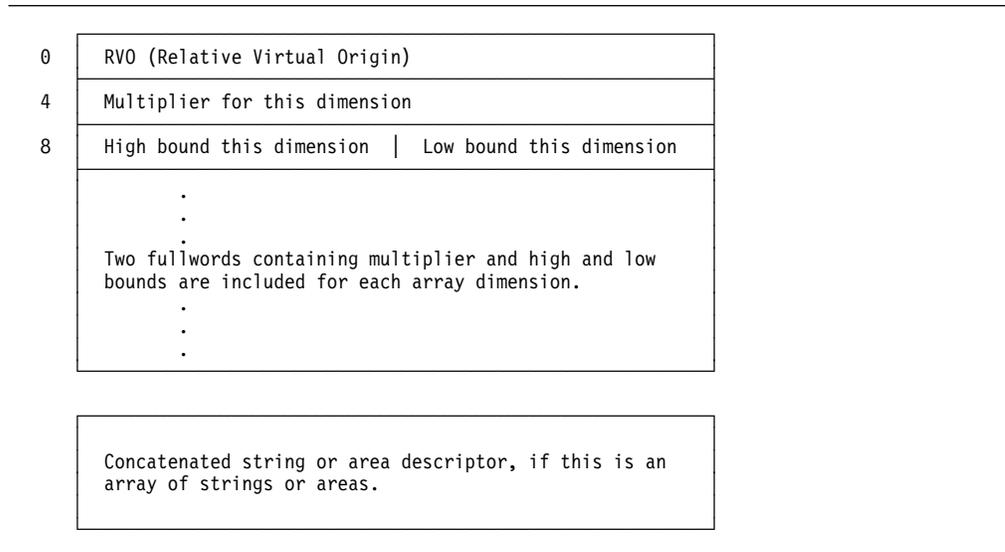


Figure 100. Format of the Array Descriptor for a Program Compiled with CMPAT(V1)

**RVO (Relative virtual origin)**

This is held as a byte value except for bit string arrays, in which case this is a bit value. For bit string arrays, the bit offset from the byte address is held in the string descriptor.

**Multiplier**

The multiplier is held as a byte value, except for bit string arrays in which case they are bit values.

**High bound**

The high subscript for this dimension

**Low bound**

The low subscript for this dimension.

## String Locator/Descriptor

The string locator/descriptor holds this information:

- The byte address of the string
- The (maximum) length of the string
- Whether or not it is a varying string
- For a bit string, the bit offset from the byte address

Figure 101 shows the format of the string locator/descriptor.

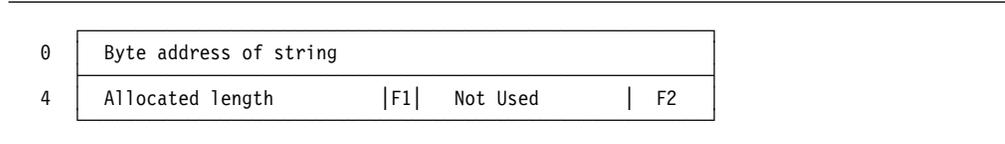


Figure 101. Format of the String Locator/Descriptor

For VARYING strings, the byte address of the string points to the half-word length prefix, which precedes the string.

Allocated length is held in bits for bit strings and in bytes for character strings. Length is held in number of graphics for graphic strings.

F1 = First bit in second halfword:  
'0'B Fixed length string  
'1'B VARYING string

F2 = Last 3 bits in second halfword:  
Used for bit strings to hold offset from byte address of  
first bit in bit string.

The string descriptor is the second word of the string locator/descriptor. It is used in structure descriptors when strings appear in structures and also in array descriptors, for arrays of strings.

## Structure Descriptor

The structure descriptor is a series of fullwords that give the byte offset of the start of each base element from the start of the structure. If a base element has a descriptor, the descriptor is included in the structure descriptor, following the appropriate fullword offset.

### Structure descriptor format:

Figure 102 shows the format of the structure descriptor. For each base element in the structure, a fullword field is present in the structure descriptor that contains the byte offset of the base element from the start of the storage for the structure. If the base element is a string, area, or array, this fullword is followed by a descriptor, which is followed by the offset field for the next base element. If the base element is not a string, area, or array, the descriptor field is omitted.

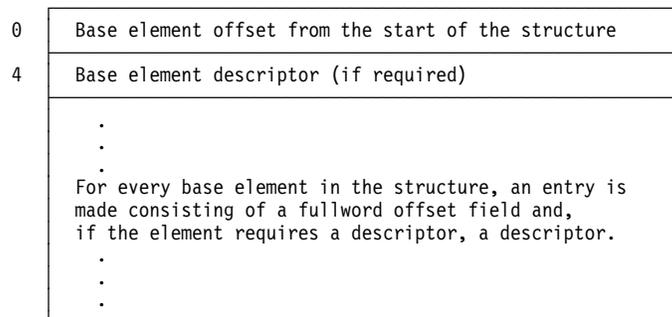


Figure 102. Format of the Structure Descriptor

Base element offsets are held in bytes. Any adjustments needed for bit-aligned addresses are held in the respective descriptors.

Major and minor structures themselves, versus the contained base elements, are not represented in the structure descriptor.

## Arrays of Structures and Structures of Arrays

When necessary, an aggregate locator and a structure descriptor are generated for both arrays of structures and structures of arrays.

The structure descriptor has the same format for both an array of structures and a structure of arrays. The difference between the two is the values in the field of the array descriptor within the structure descriptor. For example, take the array of structures AR and the structure of arrays ST, declared as follows:

<b>Array of structures</b>	<b>Structure of arrays</b>
DCL 1 AR(10),	DCL 1 ST,
2 B,	2 B(10),
2 C;	2 C(10);

The structure descriptor for both AR and ST contains an offset field and an array descriptor for B and C. However, the values in the descriptors are different, because the array of structures AR consists of elements held in the order:

B,C,B,C,B,C,B,C,B,C,B,C,B,C,B,C,B,C

but the elements in the structure of arrays ST are held in the order:

B,B,B,B,B,B,B,B,B,B,C,C,C,C,C,C,C,C,C,C.

## Chapter 17. Using PLIDUMP

This section provides information about dump options and the syntax used to call PLIDUMP, and describes PL/I-specific information included in the dump that can help you debug your routine.

**Note:** PLIDUMP conforms to National Language Support standards.

Figure 103 shows an example of a PL/I routine calling PLIDUMP to produce an Language Environment for MVS & VM dump. In this example, the main routine PLIDMP calls PLIDMPA, which then calls PLIDMPB. The call to PLIDUMP is made in routine PLIDMPB.

```

%PROCESS MAP GOSTMT SOURCE STG LIST OFFSET LC(101);
PLIDMP: PROC OPTIONS(MAIN) ;
    Declare (H,I) Fixed bin(31) Auto;
    Declare Names Char(17) Static init('Bob Teri Bo Jason');
    H = 5; I = 9;
    Put skip list('PLIDMP Starting');
    Call PLIDMPA;

    PLIDMPA: PROC;
    Declare (a,b) Fixed bin(31) Auto;
    a = 1; b = 3;
    Put skip list('PLIDMPA Starting');
    Call PLIDMPB;

    PLIDMPB: PROC;
    Declare 1 Name auto,
           2 First Char(12) Varying,
           2 Last Char(12) Varying;
    First = 'Teri';
    Last = 'Gillispy';
    Put skip list('PLIDMPB Starting');
    Call PLIDUMP('TBFC','PLIDUMP called from procedure PLIDMPB');
    Put Data;
    End PLIDMPB;

    End PLIDMPA;

End PLIDMP;

```

Figure 103. Example PL/I Routine Calling PLIDUMP

The syntax and options for PLIDUMP are shown below.

**Syntax**

```

▶▶—PLIDUMP—(—character-string-expression 1—, —————▶
▶—character-string-expression 2—)—————▶▶

```

where:

**character-string-expression 1**

is a dump options character string consisting of one or more of the following:

- A** Requests information relevant to all tasks in a multitasking program.
- B** BLOCKS (PL/I hexadecimal dump).

<b>C</b>	Continue. The routine continues after the dump.
<b>E</b>	Exit from current task of a multitasking program. Program continues to run after requested dump is completed.
<b>F</b>	FILES.
<b>H</b>	STORAGE.
	<b>Note:</b> A ddname of CEESNAP must be specified with the H option to produce a SNAP dump of a PL/I routine.
<b>K</b>	BLOCKS (when running under CICS). The Transaction Work Area is included.
<b>NB</b>	NOBLOCKS.
<b>NF</b>	NOFILES.
<b>NH</b>	NOSTORAGE.
<b>NK</b>	NOBLOCKS (when running under CICS).
<b>NT</b>	NOTRACEBACK.
<b>O</b>	Only information relevant to the current task in a multitasking program.
<b>S</b>	Stop. The enclave is terminated with a dump.
<b>T</b>	TRACEBACK.

T, F, and C are the default options.

#### **character-string-expression 2**

is a user-identified character string up to 80 characters long that is printed as the dump header.

---

## **PLIDUMP Usage Notes**

If you use PLIDUMP, the following considerations apply:

- If a routine calls PLIDUMP a number of times, use a unique user-identifier for each PLIDUMP invocation. This simplifies identifying the beginning of each dump.
- A DD statement with the ddname PLIDUMP, PL1DUMP, or CEEDUMP; a FILEDEF command in VM; or an ALLOCATE command in TSO can be used to define the data set for the dump.
- The data set defined by the PLIDUMP, PL1DUMP, or CEEDUMP DD statement should specify a logical record length (LRECL) of at least 133 to prevent dump records from wrapping.
- When you specify the H option in a call to PLIDUMP, the PL/I library issues an OS SNAP macro to obtain a dump of virtual storage. The first invocation of PLIDUMP results in a SNAP identifier of 0. For each successive invocation, the ID is increased by one to a maximum of 256, after which the ID is reset to 0.
- Support for SNAP dumps using PLIDUMP is only provided under VM and MVS. SNAP dumps are not produced in a CICS environment.

- If the SNAP is not successful, the CEE3DMP DUMP file displays the message:  
Snap was unsuccessful
- If the SNAP is successful, CEE3DMP displays the message:  
Snap was successful; snap ID = *nnn*  
where *nnn* corresponds to the SNAP identifier described above. An unsuccessful SNAP does not result in an incrementation of the identifier.

If you want to ensure portability across system platforms, use PLIDUMP to generate a dump of your PL/I routine.

---

## Chapter 18. Retaining the Run-Time Environment for Multiple Invocations

If an assembler routine is to invoke either a number of PL/I routines or the same PL/I routine repeatedly, the creation and termination of the run-time environment for each invocation will be unnecessarily inefficient. The solution is to create the run-time environment only once for use by all invocations of PL/I procedures. This can be achieved by several techniques which include:

- Preinitializing the PL/I program
- Establishing a Language Environment for MVS & VM-enabled assembler routine as the MAIN procedure
- Retaining the run-time environment using Language Environment for MVS & VM-enabled assembler as MAIN

Each of these techniques is discussed below.

---

### Preinitializable Programs

To call a PL/I program multiple times, you can establish the run-time environment and then repeatedly invoke the PL/I program using the already-established run-time environment. You incur the overhead of initializing and terminating the run-time environment only once instead of every time you invoke the PL/I program.

Because PL/I detects preinitializable programs dynamically during initialization, you do not have to recompile or relink-edit a program. However, this facility assumes that the PL/I MAIN procedure is compiled with the OS PL/I Version 2 or the PL/I for MVS & VM compiler.

**Note:** PL/I-defined preinitialization support does not include support for ILC. The support provided is slightly different from that provided by OS PL/I Version 2. Refer to the *PL/I for MVS & VM Compiler and Run-Time Migration Guide* for information about these differences.

To maintain the run-time environment, invoke the program with the PL/I entry PLISTART if you are using OS PL/I Version 2, or with CEESTART if you are using PL/I for MVS & VM, and pass a special Extended Parameter List (EPLIST) that indicates that the program is to be preinitialized.

parameter list when you pass it:

#### **INIT**

Initialize the run-time environment, return two tokens that represent the environment, but do not run the program.

#### **CALL**

Run the PL/I program using the environment established by the INIT request, and leave the environment intact when the program completes. CALL uses the two tokens that were returned by the INIT request so that PL/I can recognize the proper environment.

You can also initialize and call a PL/I program by passing the CALL parameter with two zero tokens. PL/I will service this request as an INIT followed by a CALL. You can still call the program repeatedly.

## TERM

Terminate the run-time environment but do not run the program.

## EXECUTE

Perform INIT, CALL, and TERM in succession.

In order for a program to be preinitialized it must perform the following tasks:

- Explicitly close all files, including SYSPRINT.
- Explicitly free all controlled variables.
- Explicitly release all fetched modules.

Different programs can use the same run-time environment if the programs do not use files, controlled variables, or fetched procedures. stream-oriented output to SYSPRINT, and SYSPRINT must be declared as EXTERNAL FILE.

**Note:** You cannot preinitialize a PL/I program under CICS or in conjunction with PL/I multitasking.

## The Interface for Preinitializable Programs

The interface for preinitializable programs is shown below:



The "LL" field must The following halfword must contain zeros.

The "Request" field is a field of eight characters that can contain

```
'INIT   '  
'CALL   '  
'TERM   '  
'EXECUTE'
```

No other values are allowed.

The "EPLIST" field is a pointer to the extended parameter list, which is described in the following section.

### Using the Extended Parameter List (EPLIST)

You can use the facilities made possible through the extended parameter list to support PL/I programs that run in nonproblem-state environments. These new environments include, but are not limited to:

- Supervisor state
- Cross-memory mode
- System request block (SRB) mode

## Warning

**The code that uses the preinitialization interface described in this chapter (that is, the preinitialization “director” or “driver” module) is responsible for maintaining MVS integrity.** PL/I as shipped does not prevent the operation of nonproblem-state programs in the environment described in this section, nor does it provide any means of manipulating PSW keys. Therefore, your preinitialization director code must ensure that all data areas and addresses passed as parameters to a nonproblem-state PL/I program are accessible for reading and/or writing as needed by that PL/I program in the PSW key under which the program is given control. The director code is also responsible for checking any parameters passed by the problem-state caller to ensure that the caller has the read and/or write authority needed in its PSW protect key.

The “EPLIST” field serves as a means of communication between the caller and the run-time environment. It points to the EPLIST, a vector of fullwords that The items must occur in the following order:

1. The length of the extended parameter list.
2. First token for the run-time environment.
3. Second token for the run-time environment.
4. Pointer to a parameter list Use a fullword of zeros if your code does not expect a parameter.
5. Pointer to a character string of your run-time options.

If you need them, you can optionally include the following additional fullword slots after those listed above:

6. Pointer to an alternative MAIN PL/I routine you wish to invoke. If you do not want to use this slot but wish to specify the seventh slot, use a fullword of zeros.
7. Pointer to a service vector defined by you through which you designate alternative routines to perform certain services, like loading and deleting.

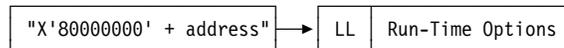
The defined constants area of the assembler module in Figure 104 on page 394 shows an example of how you could set up the extended parameter list.

The use of each field in the extended parameter list is described below.

**Length of EPLIST:** Includes the four bytes for itself. Valid decimal values are 20, 24, and 28.

**First and Second Run-Time Environment Tokens:** These tokens are automatically returned during initialization. Or, you can use zeros for them when requesting a preinitialized CALL and the effect is that both an INIT and a CALL are performed.

**Pointer to Your Program Parameters:** These parameters should be as your object code expects. If your object code expects a character string as its parameter, use the following structure:



where “LL” is a halfword containing the length of the character string.

Use a value of zero as a place holder for this slot if your program does not expect any parameters.

When parameters are passed to a preinitialized program, the parameter list must consist of one address for each parameter. The high-order bit must be turned on for the address of the last parameter.

Without knowledge of PL/I descriptors, the only data types that you can pass to a PL/I preinitialized program are pointers and numeric scalars. Numeric scalars can have any base, mode, scale, or precision.

For example, to pass the REAL FIXED BIN(31) values 5 and 6 to a preinitialized program, the parameter list could be declared as:

```

PARMS      DC  A(BIN_PARM1)
           DC  A(X'80000000'+BIN_PARM2)
BIN_PARM1  DC  F'5'
BIN_PARM2  DC  F'6'
  
```

To pass the two strings “pre” and “init” to a preinitialized program that receives them as varying strings, you should pass the parameters indirectly via pointers. The parameter list could be declared as:

```

PARMS      DC  A(PTR_PARM1)
           DC  A(X'80000000'+PTR_PARM2)
PTR_PARM1  DC  A(STR_PARM1)
PTR_PARM2  DC  A(STR_PARM2)
STR_PARM1  DC  H'3'
           DC  CL8'pre'
STR_PARM2  DC  H'4'
           DC  CL16'init'
  
```

The preinitialized PL/I program would declare its incoming parameters as pointers which could be used as locators for varying strings. The actual code might look like the following:

```

PIPISTR: Proc( String1_base, String2_base) options(main noexecops);

  Declare (String1_base, String2_base) pointer;

  Declare String1   char(08) varying based(String1_base);

  Declare String2   char(16) varying based(String2_base);
  
```

**Pointer to Your Run-Time Options:** To point to the character string of run-time options, use the structure shown in the diagram under “Pointer to Your Program Parameters” above. The run-time options provided are merged as the command-level options.

**Pointer to an Alternative Main:** If you place an address in this field and your request is for a preinitialized CALL, that address is used as the address of the compiled code to be invoked.

**Note:** When using this function, the alternative mains you invoke cannot use FETCH and RELEASE, cannot use CONTROLLED variables, and cannot use any I/O other than stream-oriented output to SYSPRINT, which must be declared as EXTERNAL FILE.

For an example of how to use this facility, see “Invoking an Alternative MAIN Routine” on page 398.

**Pointer to the Service Vector:** If you want certain services like load and delete to be carried out by some other code supplied by you (instead of, for example, the Language Environment for MVS & VM LOAD and DELETE services), you must use this field to point to the service vector. For a description of the service vector and interface requirements for user-supplied service routines, see “Using the Service Vector and Associated Routines” on page 402. Sample service routines are shown for some of the services.

**Note:** Besides interfaces and rules defined here for these services, you also might need to follow additional rules. That is, each service that you provide must follow rules defined here and rules defined for the corresponding service in the Language Environment for MVS & VM preinitialization. For detailed rules for services in Language Environment for MVS & VM-defined preinitialization, see the *Language Environment for MVS & VM Programming Guide*.

## Preinitializing a PL/I Program

Figure 104 on page 394 demonstrates how to:

- Establish the run-time environment via an INIT request
- Pass run-time parameters to the PL/I initialization routine
- Set up a parameter to the PL/I program which is different for each invocation of the program
- Repeatedly invoke a PL/I program via the CALL request
- Perform load and delete services using your own service routines instead of Language Environment for MVS & VM-provided services
- List out names of all modules loaded and deleted during the run
- Communicate from the PL/I program to the driving program via PLIRETC
- Terminate the PL/I program via the TERM request

The PL/I program itself is very simple. The parameter it expects is a fullword integer. If the parameter is the integer 3, PLIRETC is set to 555. For an example, see Figure 105 on page 398.

The assembler program which drives the PL/I program establishes the run-time environment, repeatedly invokes the PL/I program passing a parameter of 10, 9, 8, and so on. If the return code set by the PL/I program is nonzero, the assembler program terminates the run-time environment and exits.

Figure 108 on page 404 and Figure 109 on page 407 show the load and delete routines used in place of the usual Language Environment for MVS & VM-provided services. These are examples of user-supplied service routines that comply with the interface conventions discussed in “Using the Service Vector and Associated Routines” on page 402. These particular routines list out the names of all modules loaded and deleted during the run.

**Note:** This program does not include the logic that would verify the correctness of any of the invocations. This logic is imperative for proper operations.

---

```

      TITLE 'Preinit Director Module'
* * * * *
* Function: Demonstrate the use of the Preinitializable Program Facility
* Preinit Requests : INIT, CALL, TERM
*
* Parm to Prog : Fixed Bin(31)
* Output       : Return Code set via PLIRETC
*
* * * * *
NEWPIPI CSECT
        EXTRN CEESTART
        EXTRN LDMPIDI           User's load routine
        EXTRN DLMPIPI          User's delete routine
*
        DS      0H
        STM    14,12,12(13)     Save registers
        LR     R12,R15          Set module Base
        USING  NEWPIPI,R12
*
        LA     R10,SAVAREA
        ST     R10,8(R13)       Forward Chain
        ST     R13,4(R10)       Back Chain
        LR     R13,R10
*
MAINCODE EQU *
* =====
*   Setup the Request list specifying 'INIT'.
* =====
*
        WTO   'Prior to INIT request'
*
        MVC   PRP_REQUEST,INIT   Indicate the INIT request
*
        LA    R1,EXEC_ADDR       Get the parm addr list
        ST    R1,EPL_EXEC_OPTS   Save in EPL
*
* Setup R1 to point to the Parm list
*
        LA    R1,PARM_EPL        R1 --> Pointer --> Request list
        L     R15,PSTART         PL/I Entry addr
        BALR R14,R15             Invoke PL/I
*
        WTO   'After INIT request'
*

```

---

Figure 104 (Part 1 of 4). Director Module for Preinitializing a PL/I Program

```

* =====
* The run-time environment is now established. PL/I object code
* has not yet been executed. We will now repeatedly invoke
* the PL/I object code until the PL/I object code sets
* a non-zero return code.
* =====
* Setup the Request list specifying 'CALL'.
* =====
*
DO_CALL EQU *
*
*       WTO 'Prior to CALL request'
*
*       MVC PRP_REQUEST,CALL  Indicate the CALL request
*
*       LA  R1,PARMS          Get the parm addr list
*       ST  R1,EPL_PROG_PARMS Save in EPL
*
* Setup R1 to point to the Parm list
*
*       LA  R1,PARM_EPL       R1 --> Pointer --> Request list
*       L   R15,PSTART        PL/I Entry addr
*       BALR R14,R15          Invoke PL/I
*
*       LTR R15,R15           Zero PL/I return code?
*       BNZ DO_TERM           No. We are done
*
*       L   R5,PARAMETER      Change the parm ...
*       BCTR R5,0              ... passing one less ...
*       ST  R5,PARAMETER      ... each time
*
*       LTR R5,R5             Don't loop forever
*       BNZ DO_CALL
*
* =====
* The request now is 'TERM'
* =====
*
DO_TERM EQU *
*
*       ST  R15,RETCODE       SAVE PL/I RETURN CODE
*
*       WTO 'Prior to TERM request'
*
*       MVC PRP_REQUEST,TERM  Indicate a TERM command
*
*       LA  R1,0              No parm list is present
*       ST  R1,EPL_PROG_PARMS Save in EPL
*
*       LA  R1,PARM_EPL       R1 --> Pointer --> Request list
*       L   R15,PSTART        PL/I Entry addr
*       BALR R14,R15          Invoke PL/I
*
*       WTO 'After TERM request'
*

```

Figure 104 (Part 2 of 4). Director Module for Preinitializing a PL/I Program

```

=====
*   Return to the System       -       Bye
=====
SYSRET  EQU  *
*
*       L   R13,SAVAREA+4
*       L   R14,12(R13)
*       L   R15,RETCODE
*       LM  R0,R12,20(R13)
*       BR  R14           Return to your caller
*       EJECT
*       EJECT
*
*           CONSTANTS AND WORKAREAS
*
SAVAREA  DS   20F
*
RETCODE  DC   F'0'
*
PARM_EPL DC   A(X'80000000'+IBMBZPRP)   Parameter Addr List
PSTART  DC   A(CEESTART)
*
* =====
* Request strings allowed in the Interface
* =====
INIT     DC   CL8'INIT'           Initialize the program envir
CALL     DC   CL8'CALL'          Invoke the appl - leave envir up
TERM     DC   CL8'TERM'          Terminate Environment
EXEC     DC   CL8'EXECUTE'        INIT, CALL, TERM - all in one
*
* =====
*           SPACE 1
*       Parameter list passed by a Pre-Initialized Program
*       Addressed by Reg 1 = A(A(IBMBZPRP))
*       See IBMBZEPL DSECT.
*           SPACE 1
IBMBZPRP DS   0F
PRP_LENGTH DC H'16'           Len of this PRP passed (16)
PRP_ZERO   DC H'0'           Must be zero
PRP_REQUEST DC CL8' '         'INIT' - initialize PL/I
*                                     'CALL' - invoke application
*                                     'TERM' - terminate PL/I
*                                     'EXECUTE' - init, call, term
*
PRP_EPL_PTR DC A(IBMBZEPL)     A(EPL) - Extended Parm List
*
* =====
*           SPACE 1
*       Parameter list for the Pre-Initialized Programs
*           SPACE 1
IBMBZEPL  DS   0F
EPL_LENGTH DC A(EPL_SIZE)     Length of this EPL passed
EPL_TOKEN1 DC F'0'           First env token
EPL_TOKEN2 DC F'0'           Second env token
EPL_PROG_PARMS DC F'0'       A(Parm address List) ...
EPL_EXEC_OPTS DC A(EXEC_ADDR) A(Execution time optns) ...
EPL_ALTMAIN DC F'0'          A(Alternate Main)
EPL_SERVICE_VEC DC A(IBMBZSRV) A(Service Routines Vector)
EPL_SIZE   EQU *-IBMBZEPL     The size of this block
*

```

Figure 104 (Part 3 of 4). Director Module for Preinitializing a PL/I Program

```

-----
*
*      Service Routine Vector
*
IBMBZSRV          DS  0F
SRV_SLOTS         DC  F'4'          Count of slots defined
SRV_USERWORD      DC  A(SRV_UA)     user word
SRV_WORKAREA      DC  A(SRV_WA)     A(workarea)
SRV_LOAD          DC  A(LDMPIDI)    A(Load routine)
SRV_DELETE        DC  A(DLMPIDI)    A(Delete routine)
SRV_GETSTOR       DC  F'0'          A(Get storage routine)
SRV_FREESTOR      DC  F'0'          A(Free storage routine)
SRV_EXCEP_RTR     DC  F'0'          A(Exception router service)
SRV_ATTN_RTR      DC  F'0'          A(Attention router service)
SRV_MSG_RTR       DC  F'0'          A(Message router service)
SRV_END           DS  0F
*
*      Service Routine Userarea
*
SRV_UA            DS  8F
*
*      Service Routine Workarea
*
SRV_WA            DS  0D
                  DC  F'256'        Length of workarea
                  DS  63F          Actual workarea
*
-----
*      Setup the parameter to be passed to the PL/I Init and Program
*
-----
*
PARMS      DC  A(X'80000000'+PARAMETER)
PARAMETER DC  F'10'
*
EXEC_ADDR DC  A(X'80000000'+EXEC_LEN)
EXEC_LEN  DC  AL2(EXEC_OLEN)
EXEC_OPTS DC  C'NATLANG(ENU)'
EXEC_OLEN EQU *-EXEC_OPTS
*
*
      LTORG
R0      EQU  0
R1      EQU  1
R2      EQU  2
R3      EQU  3
R4      EQU  4
R5      EQU  5
R6      EQU  6
R7      EQU  7
R8      EQU  8
R9      EQU  9
R10     EQU 10
R11     EQU 11
R12     EQU 12
R13     EQU 13
R14     EQU 14
R15     EQU 15
      END

```

Figure 104 (Part 4 of 4). Director Module for Preinitializing a PL/I Program

The program in Figure 105 shows how to use the preinitializable program.

---

```
%PROCESS LIST SYSTEM(MVS) TEST;
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Function : To demonstrate the use of a preinitializable program */
/* Action   : Passed an argument of 3, it sets the PL/I return   */
/*           : code to 555.                                       */
/*                                           */
/* Input    : Fullword Integer                                     */
/* Output   : Return Code set via PLIRETC                         */
/*                                           */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
PLIEXAM: Proc (Arg) Options(MAIN);
  Dcl Sysprint File Output;
  Dcl Arg Fixed Bin(31);
  Dcl Pliretc Builtin;

  Open File(Sysprint);
  Put Skip List ('Hello' || Arg);

  If (Arg = 3) Then
    Do;
      Put Skip List ('Setting a nonzero PL/I return Code. ');
      Call Pliretc(555);
    End;

  Close File(Sysprint);

END PLIEXAM;
```

---

Figure 105. A Preinitializable Program

## Invoking an Alternative MAIN Routine

This section shows a sample director module (see Figure 106 on page 399) much like the one in the “Preinitializing a PL/I Program” on page 393, except that instead of invoking the same PL/I program repeatedly, it invokes the original program only once, and all subsequent invocations go to an alternative MAIN program.

For simplicity, the alternative MAIN (see Figure 107 on page 402) is the same program as the original one except that it has been given another name. It expects a fullword integer as its parameter, and sets PLITREC to 555 if the parameter received is the integer 3.

**Note:** When using this function, the alternative MAINS you invoke cannot use FETCH and RELEASE, cannot use CONTROLLED variables, and cannot use any I/O other than stream-oriented output to SYSPRINT, which must be declared as EXTERNAL FILE.

The following director module does not include the logic that would verify the correctness of any of the invocations. This logic is imperative for proper operations.

```

TITLE 'Preinit Director Module'
* * * * *
* Invoke PL/I program using Preinit Xaction. *
* * * * *
PIALTEPA CSECT
PIALTEPA RMODE ANY
PIALTEPA AMODE ANY
        EXTRN CEESTART
        EXTRN ALTMAIN
*
        DS    0H
        STM   14,12,12(13)    Save registers
        LR    R12,R15        Set module Base
        USING PIALTEPA,R12
*
        LA    R10,SAVAREA
        ST    R10,8(R13)     Forward Chain
        ST    R13,4(R10)    Back Chain
        LR    R13,R10
*
MAINCODE EQU *
* =====
* Setup the Request list specifying 'INIT'.
* =====
*
        WTO   'Prior to INIT request'
*
        MVC   PRP_REQUEST,INIT    Indicate the INIT request
*
        LA    R1,EXEC_ADDR        Get the parm addr list
        ST    R1,EPL_EXEC_OPTS    Save in EPL
*
* Setup R1 to point to the Parm list
*
        LA    R1,PARM_EPL        R1 --> Pointer --> Request list
        L     R15,PSTART         PL/I Entry addr
        BALR  R14,R15           Invoke PL/I
*
        WTO   'After INIT request'
*
* =====
* Setup the Request list specifying 'CALL'.
* =====
*
DO_CALL EQU *
*
        WTO   'Prior to CALL request'
*
        MVC   PRP_REQUEST,CALL    Indicate the CALL request
*
        LA    R1,PARMS           Get the parm addr list
        ST    R1,EPL_PROG_PARMS  Save in EPL
*
* Setup R1 to point to the Parm list
*
        LA    R1,PARM_EPL        R1 --> Pointer --> Request list
        L     R15,PSTART         PL/I Entry addr
        BALR  R14,R15           Invoke PL/I
*
        LTR   R15,R15           Zero PL/I return code?
        BNZ   DO_TERM           No. We are done
*
        L     R5,PARAMETER       Change the parm ...
        BCTR  R5,0              ... passing one less ...
        ST    R5,PARAMETER       ... each time

```

Figure 106 (Part 1 of 3). Director Module for Invoking an Alternative MAIN

```

* -----
*   Following code causes subsequent invocations to go to
*   alternative MAIN
* -----
      L   R15,PSTART2
      ST  R15,EPL_ALTMAIN
*
      LTR R5,R5           Don't loop forever
      BNZ DO_CALL
*
* =====
*   The request now is 'TERM'
* =====
*
DO_TERM EQU *
*
      ST  R15,RETCODE      SAVE PL/I RETURN CODE
*
      WTO 'Prior to TERM request'
*
      MVC PRP_REQUEST,TERM  Indicate a TERM command
*
      LA  R1,0             No parm list is present
      ST  R1,EPL_PROG_PARMS Save in EPL
*
      LA  R1,PARM_EPL      R1 --> Pointer --> Request list
      L   R15,PSTART       PL/I Entry addr
      BALR R14,R15         Invoke PL/I
*
      WTO 'After TERM request'
*
SYSRET EQU *
*
      L   R13,SAVAREA+4
      L   R14,12(R13)
      L   R15,RETCODE
      LM  R0,R12,20(R13)
      BR  R14              Return to your caller
      EJECT
      EJECT
*
      CONSTANTS AND WORKAREAS
*
SAVAREA DS 20F
*
RETCODE DC F'0'
PARM_EPL DC A(X'80000000'+IBMBZPRP)   Parameter Addr List
PSTART  DC A(CEESTART)
PSTART2 DC A(ALTMAIN)
* =====
* Request strings allowed in the Interface
* =====
INIT    DC CL8'INIT'      Initialize the program envir
CALL    DC CL8'CALL'      Invoke the appl - leave envir up
TERM    DC CL8'TERM'      Terminate Environment
EXEC    DC CL8'EXECUTE'   INIT, CALL, TERM - all in one
*
* =====
                                SPACE 1

```

Figure 106 (Part 2 of 3). Director Module for Invoking an Alternative MAIN

```

*      Parameter list passed by a Pre-Initialized Program
*      Addressed by Reg 1 = A(A(IBMZPRP))
*      See IBMZBEPL DSECT.
      SPACE 1
IBMZPRP      DS  0F
PRP_LENGTH   DC  H'16'      Len of this PRP passed (16)
PRP_ZERO     DC  H'0'      Must be zero
PRP_REQUEST  DC  CL8' '    'INIT' - initialize PL/I
*                                     'CALL' - invoke application
*                                     'TERM' - terminate PL/I
*                                     'EXECUTE' - init, call, term
*
PRP_EPL_PTR   DC  A(IBMZBEPL)  A(EPL) - Extended Parm List
* =====
      SPACE 1
*      Parameter list for the Pre-Initialized Programs
      SPACE 1
IBMZBEPL      DS  0F
EPL_LENGTH    DC  A(EPL_SIZE)  Length of this EPL passed
EPL_TOKEN_TCA DC  F'0'      A(TCA)
EPL_TOKEN_PRV DC  F'0'      A(PRV)
EPL_PROG_PARMS DC  F'0'      A(Parm address List) ...
EPL_EXEC_OPTS DC  A(EXEC_ADDR) A(Execution time optns) ...
EPL_ALTMAIN   DC  A(0)      A(alternate MAIN)
EPL_SERVICE_VEC DC  A(0)      A(Service Routines Vector)
EPL_SIZE      EQU  *-IBMZBEPL  The size of this block
*
*
*-----
*      Setup the parameter to be passed to the PL/I Init and Program
*-----
*
PARMS      DC  A(X'80000000'+PARAMETER)
PARAMETER DC  F'10'
*
EXEC_ADDR DC  A(X'80000000'+EXEC_LEN)
EXEC_LEN  DC  AL2(EXEC_OLEN)
EXEC_OPTS DC  C'NATLANG(ENU)'
EXEC_OLEN EQU  *-EXEC_OPTS
*
*
      LTORG
R0      EQU  0
R1      EQU  1
R2      EQU  2
R3      EQU  3
R4      EQU  4
R5      EQU  5
R6      EQU  6
R7      EQU  7
R8      EQU  8
R9      EQU  9
R10     EQU  10
R11     EQU  11
R12     EQU  12
R13     EQU  13
R14     EQU  14
R15     EQU  15
      END

```

Figure 106 (Part 3 of 3). Director Module for Invoking an Alternative MAIN

---

```

%PROCESS A(F) X(F) NIS S FLAG(I);
%PROCESS TEST(NONE,SYM);
%PROCESS SYSTEM(MVS);
  ALTMAIN: Proc (Arg) options (main);
  Dcl Arg Fixed Bin(31);
  Dcl Pliretc Builtin;
  Dcl Oncode Builtin;
  Dcl Sysprint File Output Stream;

  On Error Begin;
    On error system;
    Put Skip List ('ALTMAIN - In error On-unit. OnCode = '||Oncode );
  End;

  Put Skip List ('ALTMAIN - Entering Pli');
  Put Skip List (' Arg = ' || Arg );

  If Arg = 3 Then
    do;
      Put Skip List ('Setting a nonzero PL/I return code');
      Call Pliretc(555);
    end;
  Else;

  Put skip List ('ALTMAIN - Leaving Pli');

END;

```

---

Figure 107. Alternative MAIN Routine

## Using the Service Vector and Associated Routines

This section describes the service vector, which is a list of addresses of various user-supplied service routines. Also described are the interface requirements for each of the service routines that you can supply, including sample routines for some of the services.

**Note:** These services must be AMODE(ANY) and RMODE(24). You must also follow further rules defined for services that Language Environment for MVS & VM-defined preinitialization provides.

### Using the Service Vector

If you want certain services like load and delete to be carried out by some other code supplied by you (instead of, for example, the Language Environment for MVS & VM LOAD and DELETE services), you must place the address of your service vector in the seventh fullword slot of the extended parameter list (see “Using the Extended Parameter List (EPLIST)” on page 390). You must define the service vector according to the pattern shown in the following example:

SRV_COUNT	DS	F	Count of slots defined
SRV_USER_WORD	DS	F	User-defined word
SRV_WORKAREA	DS	A	Addr of work area for dsas etc
SRV_LOAD	DS	A	Addr of load routine
SRV_DELETE	DS	A	Addr of delete routine
SRV_GETSTOR	DS	A	Addr of get-storage routine
SRV_FREESTOR	DS	A	Addr of free-storage routine
SRV_EXCEP_RTR	DS	A	Addr of exception router
SRV_ATTN_RTR	DS	A	Addr of attention router
SRV_MSG_RTR	DS	A	Addr of message router

Although you do not need to use labels identical to those above, you must use the order above. That is, the address of your free-storage routine is seventh, the address of your load routine is fourth, and so on.

Some other constraints apply:

- When you define the service vector, you must fill all the slots in the template that precede the last one you specify. You can, however, supply zeros for the slots that you want ignored.
- The slot count does not count itself. The maximum value is therefore 9.
- You must specify an address in the work area slot if you specify addresses in any of the subsequent fields.
- This work area must begin on a doubleword boundary and start with a fullword that specifies its length. The length must be at least 256 bytes.
- For the load and delete routines, you cannot specify one of the pair without the other; if one of these two slots contains a value of zero, the other is automatically ignored.
- For the get-storage and free-storage routines, you cannot specify one of the pair without the other; if one of these two slots contains a value of zero, the other is automatically ignored.
- If you specify the get- and free-storage services, you must also specify the load and delete services.

For an example of a service vector definition, see the defined constants area of the assembler module in Figure 104 on page 394.

You are responsible for supplying any service routines pointed to in your service vector. These service routines, upon invocation, can expect:

- Register 13 to point to a standard 18-fullword save area
- Register 1 to point to a list of addresses of parameters available to the routine
- The third parameter in the list to be the address of the user word you specified in the second slot of the service vector

The parameters available to each routine, and the return and reason codes each routine is expected to use are given in the following sections. The parameter addresses are passed in the same order as the order in which the parameters are listed.

### **Load Service Routine**

The load routine is responsible for loading named modules. This is the service typically obtained via the LOAD macro.

The parameters passed to the load routine are:

Parameter	PL/I attributes	Type
Address of module name	POINTER	Input
Length of name	FIXED BIN(31)	Input
User word	POINTER	Input
(Reserved field)	FIXED BIN(31)	Input
Address of load point	POINTER	Output
Size of module	FIXED BIN(31)	Output
Return code	FIXED BIN(31)	Output
Reason code	FIXED BIN(31)	Output

The name length must not be zero. You can ignore the reserved field: it will contain zeros. For additional rules about the LOAD service, see the *Language Environment for MVS & VM Programming Guide*.

The return/reason codes that the load routine can set are:

- 0/0     successful
- 0/4     successful—found as a VM nucleus extension
- 0/8     successful—loaded as a VM shared segment
- 0/12    successful—loaded using SVC8
- 4/4     unsuccessful—module loaded above line when in AMODE(24)
- 8/4     unsuccessful—load failed
- 16/4    unsuccessful—uncorrectable error occurred

Figure 108 shows a sample load routine that displays the name of each module as it is loaded.

```

TITLE 'Preinit Load Service Routine'
* * * * *
* Load service routine to be used by preinit under MVS *
* * * * *
LDMPIPI CSECT
*
    DS    0H
    STM   14,12,12(13)    Save registers
    LR    R3,R15         Set module Base
    USING LDMPIPI,R3
*
    LR    R2,R1          Save parm register
    USING SRV_LOAD_PLIST,R2
    USING SRV_PARM,R4

```

Figure 108 (Part 1 of 3). User-Defined Load Routine

```

*
L   R4,SRV_LOAD_A_USERWORD Get a(userword)
L   R1,SRV_PARM_VALUE      Get userword
MVC 0(12,R1),WTOCTL        Move in WTO skeleton
L   R4,SRV_LOAD_A_NAME     Get a(a(module name))
L   R15,SRV_PARM_VALUE     Get a(module name)
* the following assumes the name is 8 characters
MVC 12(8,R1),0(R15)       Move in name
SVC 35

*
L   R4,SRV_LOAD_A_NAME     Get a(a(module name))
L   R0,SRV_PARM_VALUE     Get a(module name)

*
LOAD EPLOC=(0)

*
L   R4,SRV_LOAD_A_LOADPT  Get a(a(loadpt))
ST  R0,SRV_PARM_VALUE     Set a(module) in parmlist
L   R4,SRV_LOAD_A_SIZE    Get a(a(size))
ST  R1,SRV_PARM_VALUE     Set l(module) in parmlist
SR  R0,R0                 Get zero for codes
L   R4,SRV_LOAD_A_RETCODE Get a(retcode)
ST  R0,SRV_PARM_VALUE     Set retcode = 0
L   R4,SRV_LOAD_A_RSNCODE Get a(rsncode)
ST  R0,SRV_PARM_VALUE     Set rsncode = 0

*
LM  R14,R12,12(R13)
BR  R14                   Return to your caller

*

*
WTOCTL DS  0H
WTOWLEN DC  AL2(WTOEND-WTOCTL)
WTOFLG DC  X'8000'
WTOAREA DC  CL8'LOAD'
WTONAME DS  CL8
WTOEND DS  0X
WTOLEN EQU *-WTOCTL
EJECT

* =====
*
*   Declare to de-reference parameter pointers
*
*
SRV_PARM          SPACE 1
SRV_PARM_VALUE   DSECT
SRV_PARM_VALUE   DS  A           Parameter value
*
*

```

Figure 108 (Part 2 of 3). User-Defined Load Routine

---

```

*      Parameter list for LOAD service
*
          SPACE 1
SRV_LOAD_PLIST      DSECT
SRV_LOAD_A_NAME     DS  A          A(A(module name))
SRV_LOAD_A_NAMELEN  DS  A          A(Length of name)
SRV_LOAD_A_USERWORD DS  A          A(User word)
SRV_LOAD_A_RSVD     DS  A          A(Reserved - must be 0)
SRV_LOAD_A_LOADPT   DS  A          A(A(module load point))
SRV_LOAD_A_SIZE     DS  A          A(Size of module)
SRV_LOAD_A_RETCODE  DS  A          A(Return code)
SRV_LOAD_A_RSNCODE  DS  A          A(Reason code)
*
*
*      LTORG
R0      EQU  0
R1      EQU  1
R2      EQU  2
R3      EQU  3
R4      EQU  4
R5      EQU  5
R6      EQU  6
R7      EQU  7
R8      EQU  8
R9      EQU  9
R10     EQU 10
R11     EQU 11
R12     EQU 12
R13     EQU 13
R14     EQU 14
R15     EQU 15
          END

```

---

Figure 108 (Part 3 of 3). User-Defined Load Routine

## Delete Service Routine

The delete routine is responsible for deleting named modules. This is the service typically obtained via the DELETE macro.

The parameters passed to the delete routine are:

Parameter	PL/I attributes	Type
Address of module name	POINTER	Input
Length of name	FIXED BIN(31)	Input
User word	POINTER	Input
(Reserved field)	FIXED BIN(31)	Input
Return code	FIXED BIN(31)	Output
Reason code	FIXED BIN(31)	Output

The name length must not be zero. You can ignore the reserved field: it will contain zeros. Every delete must have a corresponding load, and the task that does the load must also do the delete. Counts of deletes and loads performed must be maintained by the service routines themselves.

The return/reason codes that the delete routine can set are:

```

0/0      successful
8/4      unsuccessful—delete failed
16/4     uncorrectable error occurred

```

Figure 109 on page 407 shows a sample delete routine that displays the name of each module as it is deleted.

```

TITLE 'Preinit Delete Service Routine'
* * * * *
* Delete service routine to be used by preinit under MVS *
* * * * *
DLMPIPI CSECT
*
DS 0H
STM 14,12,12(13) Save registers
LR R3,R15 Set module Base
USING DLMPIPI,R3
*
LR R2,R1 Save parm register
USING SRV_DELE_PLIST,R2
USING SRV_PARM,R4
*
L R4,SRV_DELE_A_USERWORD Get a(userword)
L R1,SRV_PARM_VALUE Get userword
MVC 0(12,R1),WTOCTL Move in WTO skeleton
L R4,SRV_DELE_A_NAME Get a(a(module name))
L R15,SRV_PARM_VALUE Get a(module name)
* the following assumes the name is 8 characters
MVC 12(8,R1),0(R15) Move in name
SVC 35
*
L R4,SRV_DELE_A_NAME Get a(a(module name))
L R0,SRV_PARM_VALUE Get a(module name)
*
DELETE EPLOC=(0)
*
SR R0,R0 Get zero for codes
L R4,SRV_DELE_A_RETCODE Get a(retcode)
ST R0,SRV_PARM_VALUE Set retcode = 0
L R4,SRV_DELE_A_RSNCODE Get a(rsncode)
ST R0,SRV_PARM_VALUE Set rsncode = 0
*
LM R14,R12,12(R13)
BR R14 Return to your caller
*
*
WTOCTL DS 0H
WTOWLEN DC AL2(WTOEND-WTOCTL)
WTOFLG DC X'8000'
WTOAREA DC CL8'DELETE'
WTONAME DS CL8
WTOEND DS 0X
WTOLEN EQU *-WTOCTL
EJECT
* =====
*
* Declare to dereference parameter pointers
*
SRV_PARM SPACE 1
SRV_PARM DSECT
SRV_PARM_VALUE DS A Parameter value
*

```

Figure 109 (Part 1 of 2). User-Defined Delete Routine

```

*
*   Parameter list for DELETE service
*
*
*           SPACE 1
SRV_DELE_PLIST      DSECT
SRV_DELE_A_NAME     DS  A           A(A(module name))
SRV_DELE_A_NAMELEN  DS  A           A(Length of name)
SRV_DELE_A_USERWORD DS  A           A(User word)
SRV_DELE_A_RSVD     DS  A           A(Reserved - must be 0)
SRV_DELE_A_RETCODE  DS  A           A(Return code)
SRV_DELE_A_RSNCODE  DS  A           A(Reason code)
*
*
*
*           LTORG
R0      EQU  0
R1      EQU  1
R2      EQU  2
R3      EQU  3
R4      EQU  4
R5      EQU  5
R6      EQU  6
R7      EQU  7
R8      EQU  8
R9      EQU  9
R10     EQU 10
R11     EQU 11
R12     EQU 12
R13     EQU 13
R14     EQU 14
R15     EQU 15
END

```

Figure 109 (Part 2 of 2). User-Defined Delete Routine

## Get-Storage Service Routine

The get-storage routine is responsible for obtaining storage. This is the service typically obtained via the GETMAIN, DMSFREE, or CMSSTOR macros.

The parameters passed to the get-storage routine are:

Parameter	PL/I attributes	Type
Amount desired	FIXED BIN(31)	Input
Subpool number	FIXED BIN(31)	input
User word	POINTER	Input
Flags	BIT(32)	Input
Address of gotten storage	POINTER	Output
Amount obtained	FIXED BIN(31)	Output
Return code	FIXED BIN(31)	Output
Reason code	FIXED BIN(31)	Output

The return/reason codes for the get-storage routine are:

```

0/0      successful
16/4     unsuccessful—uncorrectable error occurred

```

All storage must be obtained conditionally by the service routine. Bit 0 in the flags is on if storage is wanted below the line. The other bits are reserved and might or might not be zero.

## Free-Storage Service Routine

The free-storage routine is responsible for freeing storage. This is the service typically obtained via the FREEMAIN, DMSFRET, or CMSSTOR macros.

The parameters passed to the free-storage routine are:

Parameter	PL/I attributes	Type
Amount to be freed	FIXED BIN(31)	Input
Subpool number	FIXED BIN(31)	Input
User word	POINTER	Input
Address of storage	POINTER	Input
Return code	FIXED BIN(31)	Output
Reason code	FIXED BIN(31)	Output

The return/reason codes for the free-storage service routine are:

0/0      successful  
16/0     unsuccessful—uncorrectable error occurred

## Exception Router Service Routine

The exception router is responsible for trapping and routing exceptions. These are the services typically obtained via the ESTAE and ESPIE macros.

The parameters passed to the exception router are:

Parameter	PL/I attributes	Type
Address of exception handler	POINTER	Input
Environment token	POINTER	Input
User word	POINTER	Input
Abend flags	BIT(32)	Input
Check flags	BIT(32)	Input
Return code	FIXED BIN(31)	Output
Reason code	FIXED BIN(31)	Output

*During initialization*, if the TRAP(ON) option is in effect, the Language Environment for MVS & VM library puts the address of the common library exception handler in the first field of the above parameter list, and sets the environment token field to a value that will be passed on to the exception handler. It also sets abend and check flags as appropriate, and then calls your exception router to establish an exception handler.

The meaning of the bits in the abend flags are given by the following declare:

```

dcl
  1 abendflags,
    2 system,
      3 abends bit(1), /* control for system abends desired */
      3 rsrv1 bit(15), /* reserved */
    2 user,
      3 abends bit(1), /* control for user abends desired */
      3 rsrv2 bit(15); /* reserved */

```

The meaning of the bits in the check flags is given by the following declare:

```

1 checkflags,
  2 type,
    3 reserved3 bit(1),
    3 operation bit(1),
    3 privileged_operation bit(1),
    3 execute bit(1),
    3 protection bit(1),
    3 addressing bit(1),
    3 specification bit(1),
    3 data bit(1),
    3 fixed_overflow bit(1),
    3 fixed_divide bit(1),
    3 decimal_overflow bit(1),
    3 decimal_divide bit(1),
    3 exponent_overflow bit(1),
    3 exponent_underflow bit(1),
    3 significance bit(1),
    3 float_divide bit(1),
  2 reserved4 bit(16);

```

The return/reason codes that the exception router must use are:

```

0/0    successful
4/4    unsuccessful—the exit could not be (de-)established
16/4   unrecoverable error occurred

```

*When an exception occurs*, the exception router must determine if the established exception handler is interested in the exception (by examining abend and check flags). If the exception handler *is not* interested in the exception, the exception router must treat the program as in error, but can assume the environment for the thread to be functional and reusable. If the exception handler *is* interested in the exception, the exception router must invoke the exception handler, passing the following parameters:

Parameter	PL/I attributes	Type
Environment token	POINTER	Input
SDWA	POINTER	Input
Return code	FIXED BIN(31)	Output
Reason code	FIXED BIN(31)	Output

The return/reason codes upon return from the exception handler are as follows:

- 0/0 Continue with the exception. Percolate the exception taking whatever action would have been taken had it not been handled at all. In this case, your exception router can assume the environment for the thread to be functional and reusable.
- 0/4 Continue with the exception. Percolate the exception taking whatever action would have been taken had it not been handled at all. In this case, the environment for the thread is probably unreliable and not reusable. A forced termination is suggested.
- 4/0 Resume execution using the updated SDWA. The invoked exception handler will have already used the SETRP RTM macro to set the SDWA for correct resumption.

*During termination*, your exception router is invoked with the exception handler address (first parameter) set to zero in order to de-establish the exit—if it was established during initialization.

For certain exceptions, the common library exception handler calls your exception router to establish another exception handler exit, and then makes a call to de-establish it before completing processing of the exception. If an exception occurs while the second exit is active, special processing is performed. Depending on what this second exception is, either the first exception will not be retried, or processing will continue on the first exception by requesting retry for the second exception.

If the common library exception handler determines that execution should resume for an exception, it will set the SDWA with SETRP and return with return/reason codes 4/0. Execution will resume in library code or in user code, depending on what the exception was.

Your exception router must be capable of restoring all the registers from the SDWA when control is given to the retry routine. This is required for PL/I to resume execution at the point where an interrupt occurred for the ZERODIVIDE, OVERFLOW, FIXEDOVERFLOW, and UNDERFLOW conditions, and certain instances of the SIZE condition. The ESPIE and ESTAE services are capable of accomplishing this. The SETFRR service will not restore register 15 to the requested value. If you cannot restore all the registers, unpredictable results can occur for a normal return from an ON-unit for one of these conditions. Unpredictable results can also occur if any of these conditions are disabled by a condition prefix. In addition, unpredictable results can occur for the implicit action for the UNDERFLOW condition.

To avoid unpredictable results in PL/I programs when not all of the registers can be restored, you must code an ON-unit for UNDERFLOW that does not return normally. For the other conditions, the implicit action is to raise the error condition, which is a predictable result. If you use an ON-unit for one of these conditions, you should guarantee that the ON-unit will not return normally.

In using the exception router service, you should also be aware that:

- Your exception router should not invoke the Language Environment for MVS & VM library exception handler if active I/O has been halted and is not restorable.

- You cannot invoke any Language Environment for MVS & VM dump services including PLIDUMP.
- This service requires an ESA\* environment.
- This service is not supported under VM.

If an exception occurs while the exception handler is in control before another exception handler exit has been stacked, your exception router should assume that exception could not be handled and that the environment for the program (thread) is damaged. In this case, your exception router should force termination of the preinitialized environment.

### Attention Router Service Routine

The attention router is responsible for trapping and routing attention interrupts. These are the services typically obtained via the STAX macro.

The parameters passed to the attention router are:

Parameter	PL/I attributes	Type
Address of attention handler	POINTER	Input
Environment token	POINTER	Input
User word	POINTER	Input
Return code	FIXED BIN(31)	Output
Reason code	FIXED BIN(31)	Output

The return/reason codes upon return from the attention router are as follows:

0/0      successful  
 4/4      unsuccessful—the exit could not be (de-)established  
 16/4     unrecoverable error occurred

When an attention interrupt occurs, your attention router must invoke the attention handler using the address in the attention handler field, passing the following parameters:

Parameter	PL/I attributes	Type
Environment token	POINTER	Input
Return code	FIXED BIN(31)	Output
Reason code	FIXED BIN(31)	Output

The return/reason codes upon return from the attention handler are as follows:

0/0      attention has been (will be) handled

If an attention interrupt occurs while in the attention handler or when an attention handler is not established at all, your attention router should ignore the attention interrupt.

Figure 110 on page 413 shows a sample attention router.

```

TITLE 'Preinit Attention Router Service Routine'
* * * * *
* Attention service routine to be used by preinit *
* * * * *
ATAPIPI CSECT
*
DS 0H
STM 14,12,12(13) Save registers
LR R3,R15 Set module Base
USING ATAPIPI,R3
*
LR R2,R1 Save parm register
USING SRV_ATTN_PLIST,R2
USING SRV_PARM,R4
*
L R4,SRV_ATTN_A_USERWORD Get A(userword)
L R11,SRV_PARM_VALUE Get userword -- storage area
USING ZSA,R11
WTO 'In attn router'
*
L R4,SRV_ATTN_A_HANDLER Get a(a(handler))
ICM R5,X'F',SRV_PARM_VALUE Get a(handler)
BZ CANCEL_STAX A(0) means cancel
*
L R4,SRV_ATTN_A_TOKEN Get a(a(token))
L R6,SRV_PARM_VALUE Get a(token)
*
ST R5,ZHANDLER Set handler in user parm list
ST R6,ZTOKEN Set token in user parm list
LA R5,ZHANDLER Address user parm list for STAX
*
LA R4,RETPOINT Address attention exit routine
*
XC ZSTXLST(STXLTH),ZSTXLST Clear STAX work area
*
* Set appropriate SPLEVEL
*
SPLEVEL SET=2 Get XA or ESA version of macro
STAX (R4),REPLACE=YES, C
USADDR=(R5),MF=(E,ZSTXLST)
*
B EXIT_CODE
*
CANCEL_STAX DS 0H
*
STAX
*
EXIT_CODE DS 0H
*
SR R0,R0 Get zero for codes
L R4,SRV_ATTN_A_RETCODE Get a(retcode)
ST R0,SRV_PARM_VALUE Set retcode = 0
L R4,SRV_ATTN_A_RSNCODE Get a(rsncode)
ST R0,SRV_PARM_VALUE Set rsncode = 0
*
LM R14,R12,12(R13)
BR R14 Return to your caller
EJECT

```

Figure 110 (Part 1 of 4). User-Defined Attention Router

```

*
RETPOINT DS    0H
*
*   This is the attention exit routine, entered after an
*   attention interrupt.
*
        STM    14,12,12(13)      Save registers
        BALR   R3,0              Set module base
        USING *,R3
        LR     R2,R1             Save parm register
*
        WTO   'In stax exit'
*
        USING ZAXP,R2
        L     R11,ZAUSA          Get A(User area) -- our storage
*
        L     R14,ZTOKEN         Get A(token)
        ST    R14,SRV_AHND_TOKEN Set in its place
        LA    R14,SRV_AHND_TOKEN Get...
        LA    R15,SRV_AHND_RETCODE ..addresses...
        LA    R0,SRV_AHND_RSNCODE ..of parms
        STM   R14,R0,SRV_AHND_A_TOKEN Set parms for handler
*
        LA    R14,ZSSA          Get A(save area)
        ST    R13,4(,R14)       Set chain
        LR    R13,R14          Set new save area
*
        LA    R1,SRV_AHND_PLIST  Reg 1 to parameter list
        L     R15,ZHANDLER       Get A(handler)
        BALR  R14,R15           Invoke PL/I's handler
*
        L     R13,4(,R13)       Restore save area
*
        LM    R14,R12,12(R13)
        BR    R14              Return to your caller
*
*
*
ZSA      DSECT
* user address points to here on STAX invocation
ZHANDLER DS    F                A(handler)
ZTOKEN   DS    F                A(token)
* end of information needed when Attention occurs
ZSSA     DS    18F              Save area
ZSTXLST  DS    0F              STAX Plist
        SPLEVEL SET=2          Get XA or ESA version of macro
        STAX  0,MF=L
        DS    0D                ALIGN
STXLTH   EQU   *-ZSTXLST       LENGTH
*

```

Figure 110 (Part 2 of 4). User-Defined Attention Router

---

```

*      Parameter list for Attention Handler routine
*
SRV_AHND_PLIST      DS  0F
SRV_AHND_A_TOKEN   DS  A      A(Token)
SRV_AHND_A_RETCODE DS  A      A(Return code)
SRV_AHND_A_RSNCODE DS  A      A(Reason code)
*      end of parameter list
SRV_AHND_TOKEN     DS  A      Token
SRV_AHND_RETCODE   DS  A      Return code
SRV_AHND_RSNCODE   DS  A      Reason code
*
* =====
*
ZAXP      DSECT      Attention exit Plist
ZATAIE    DS    F      A(Terminal attention
*              interrupt element - TAIE)
ZAIBUF    DS    F      A(Input buffer)
ZAUSA     DS    F      A(User area)
*
*      Declare to dereference parameter pointers
*
*              SPACE 1
SRV_PARM      DSECT
SRV_PARM_VALUE DS  A      Parameter value
*
*
*      Parameter list for Attention Router service
*
*              SPACE 1
SRV_ATTN_PLIST DSECT
SRV_ATTN_A_HANDLER DS  A      A(Handler)
SRV_ATTN_A_TOKEN   DS  A      A(Token)
SRV_ATTN_A_USERWORD DS  A      A(User word)
SRV_ATTN_A_RETCODE DS  A      A(Return code)
SRV_ATTN_A_RSNCODE DS  A      A(Reason code)
SRV_ATTN_HANDLER   DS  A      Handler
SRV_ATTN_TOKEN     DS  A      Token
SRV_ATTN_USERWORD  DS  A      User word
SRV_ATTN_RETCODE   DS  A      Return code
SRV_ATTN_RSNCODE   DS  A      Reason code
*

```

---

Figure 110 (Part 3 of 4). User-Defined Attention Router

---

```

*
*
      LTORG
R0      EQU  0
R1      EQU  1
R2      EQU  2
R3      EQU  3
R4      EQU  4
R5      EQU  5
R6      EQU  6
R7      EQU  7
R8      EQU  8
R9      EQU  9
R10     EQU 10
R11     EQU 11
R12     EQU 12
R13     EQU 13
R14     EQU 14
R15     EQU 15
      END

```

---

Figure 110 (Part 4 of 4). User-Defined Attention Router

### Message Router Service Routine

The message router is responsible for routing messages generated during execution. This is the service typically obtained via the WTO and other macros.

The parameters passed to the message router are:

Parameter	PL/I attributes	Type
Address of message	POINTER	Input
Message length in bytes	FIXED BIN(31)	Input
User word	POINTER	Input
Line length	FIXED BIN(31)	Output
Return code	FIXED BIN(31)	Output
Reason code	FIXED BIN(31)	Output

If the address of the message is zero, your message router is expected to return the size of the line to which messages are written (in the line length field). This allows messages to be formatted correctly—that is, broken at blanks, etc.

The return/reason codes that the message router must use are as follows:

```

0/0      successful
16/4     unrecoverable error occurred

```

Figure 111 on page 417 shows a sample message router.

```

TITLE 'Preinit Message Service Routine'
* * * * *
* Message service routine to be used by preinit *
* * * * *
MSGPIPI CSECT
*
DS 0H
STM 14,12,12(13) Save registers
LR R3,R15 Set module Base
USING MSGPIPI,R3
*
LR R2,R1 Save parm register
USING SRV_MSG_PLIST,R2
USING SRV_PARM,R4
*
L R4,SRV_MSG_A_ADDR Get a(a(message))
L R5,SRV_PARM_VALUE Get a(message)
*
* If the address of the message is zero then
* return the message line length (the maximum
* length a message should be)
*
LTR R5,R5 a(message) zero?
BNZ WRT_MSG if not, go write msg
*
LA R5,72 buffer length = 72
L R4,SRV_MSG_A_LRECL set message lrecl = 72
ST R5,SRV_PARM_VALUE
B DONE
*
WRT_MSG DS 0H
L R4,SRV_MSG_A_USERWORD Get a(userword)
L R1,SRV_PARM_VALUE Get userword
MVC 0(12,R1),WTOCTL Move in WTO skeleton
MVI 13(R1),C' ' Blank out the
MVC 14(71,R1),13(R1) message buffer
*
L R4,SRV_MSG_A_ADDR Get a(a(message))
L R5,SRV_PARM_VALUE Get a(message)
L R4,SRV_MSG_A_LEN Get a(message length)
L R6,SRV_PARM_VALUE Get message length
BCTR R6,0 subtract 1 from length for EX
EX R6,MOVE Move in the message
*
SVC 35 Use WTO to display message
*
DONE DS 0H
SR R0,R0 Get zero for codes
L R4,SRV_MSG_A_RETCODE Get a(retcode)
ST R0,SRV_PARM_VALUE Set retcode = 0
L R4,SRV_MSG_A_RSNCODE Get a(rsncode)
ST R0,SRV_PARM_VALUE Set rsncode = 0
*
LM R14,R12,12(R13)
BR R14 Return to your caller
*

```

Figure 111 (Part 1 of 2). User-Defined Message Router

```

MOVE      MVC  12(0,R1),0(R5)
WTOCTL   DS   0H
WTOVLN   DC   AL2(WTOEND-WTOCTL)
WTOFLG   DC   X'8000'
WTOAREA  DC   CL8'MESSAGE'
WTOMSG   DS   CL72
WTOEND   DS   0X
WTOLEN   EQU  *-WTOCTL
          EJECT
* =====
*
*   Declare to dereference parameter pointers
*
          SPACE 1
SRV_PARM  DSECT
SRV_PARM_VALUE DS A          Parameter value
*
*
*   Parameter list for Message service
*
          SPACE 1
SRV_MSG_PLIST DSECT
SRV_MSG_A_ADDR DS A          A(A(message))
SRV_MSG_A_LEN  DS A          A(message length)
SRV_MSG_A_USERWORD DS A      A(User word)
SRV_MSG_A_LRECL DS A        A(line length)
SRV_MSG_A_RETCODE DS A      A(Return code)
SRV_MSG_A_RSNCODE DS A      A(Reason code)
*
*
*   LTOrg
*
LTOrg
R0      EQU  0
R1      EQU  1
R2      EQU  2
R3      EQU  3
R4      EQU  4
R5      EQU  5
R6      EQU  6
R7      EQU  7
R8      EQU  8
R9      EQU  9
R10     EQU 10
R11     EQU 11
R12     EQU 12
R13     EQU 13
R14     EQU 14
R15     EQU 15
          END

```

Figure 111 (Part 2 of 2). User-Defined Message Router

## User Exits in Preinitializable Programs

The user exits are invoked when initialization and termination is actually performed according to the user exit invocation rules defined by Language Environment for MVS & VM. That is, the user exit is invoked for initialization during the INIT request or the CALL with the zero token request. Similarly, the user exit is called for termination only during the TERM request.

## The SYSTEM Option in Preinitializable Programs

PL/I honors the SYSTEM compile-time option when evaluating the object code's parameter, except for the SYSTEM(CMS) and SYSTEM(CMSTPL) options. To invoke a preinitialized program under VM, the program **must** have been compiled using the SYSTEM(MVS) compile-time option. **Preinitializable programs are not allowed for programs compiled with SYSTEM(CMS) or SYSTEM(CMSTPL).**

For the remaining SYSTEM options (MVS, CICS, TSO, and IMS), PL/I will interpret the object code's parameter list as dictated by the SYSTEM option.

**Note:** PL/I-defined preinitialization is not supported under CICS.

To preinitialize programs in the VM environment, you must compile with the SYSTEM(MVS) option and drive the PL/I program via an assembler program.

## Calling a Preinitializable Program under VM

The following series of VM commands runs programs shown in Figure 104 on page 394 and Figure 105 on page 398:

---

```
PLIOPT PLIEXAM ( SYSTEM ( MVS
HASM PREPEXAM
LOAD PREPEXAM PLIEXAM ( RESET PREPEXAM
START *
```

---

*Figure 112. Commands for Calling a Preinitializable PL/I Program under VM*

## Calling a Preinitializable Program under MVS

Figure 113 on page 420 is a skeleton JCL which runs the example of a preinitializable program in an MVS environment.

```

//PREINIT JOB
//* * * * *
//* Assemble the driving assembler program
//* * * * *
//ASM EXEC PGM=IEV90,PARM='OBJECT,NODECK'
//SYSPRINT DD SYSOUT=A
//SYSLIB DD DSN=SYS1.MACLIB,DISP=SHR
//SYSLIN DD DSN=&&OBJ1,DISP=(,PASS),UNIT=SYSDA,
// DCB=(RECFM=FB,LRECL=80,BLKSIZE=3120),SPACE=(CYL,(2,1))
//SYSUT1 DD DSN=&&SYSUT1,UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSIN DD *
***** ASSEMBLER PROGRAM GOES HERE *****

//* * * * *
//* Compile the PL/I program
//* * * * *

//PLI EXEC PGM=IEL1AA,PARM='OBJECT,NODECK',REGION=512K
//STEPLIB DD DSN=IEL.V1R1M1.SIELCOMP,DISP=SHR
// DD DSN=CEE.V1R2M0.SCEERUN,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN DD DSN=&&LOADSET,DISP=(MOD,PASS),UNIT=SYSDA,
// SPACE=(80,(250,100)),DCB=(BLKSIZE=3120)
//SYSUT1 DD DSN=&&SYSUT1,UNIT=SYSDA,
// SPACE=(1024,(200,50)),,CONTIG,ROUND),DCB=BLKSIZE=1024
//SYSIN DD *
***** PLIEXAM PLIOPT PROGRAM GOES HERE *****

//*****
//* Link-edit the program
//*****
//LKED EXEC PGM=IEWL,PARM='XREF,LIST',COND=(9,LT,PLI),REGION=512K
//SYSLIB DD DSN=CEE.V1R2M0.SCEELKED,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN DD DSN=&&OBJ1,DISP=(OLD,DELETE)
// DD DSN=&&LOADSET,DISP=(OLD,DELETE)
// DD *
// ENTRY name of preinit csect
//SYSLMOD DD DSN=&&GOSET(GOPGM),DISP=(MOD,PASS),UNIT=SYSDA,
// SPACE=(1024,(50,20,1))
//SYSUT1 DD DSN=&&SYSUT1,UNIT=SYSDA,SPACE=(1024,(200,20)),
// DCB=BLKSIZE=1024
//*****
//* Execute the preinitializable program
//*****
//GO EXEC PGM=*.LKED.SYSLMOD,COND=((9,LT,PLI),(9,LT,LKED)),
// REGION=2048K
//STEPLIB DD DSN=CEE.V1R2M0.SCEERUN,DISP=SHR
//SYSPRINT DD SYSOUT=*
//CEEDUMP DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*

```

Figure 113. JCL for Running a Preinitializable PL/I Program under MVS

---

## **Establishing an Language Environment for MVS & VM-Enabled Assembler Routine as the MAIN Procedure**

For information about compatibility of an assembler routine as the MAIN procedure, see the *PL/I for MVS & VM Compiler and Run-Time Migration Guide*.

---

## **Retaining the Run-Time Environment Using Language Environment for MVS & VM-Enabled Assembler as MAIN**

For information about retaining the environment for better performance, see the *Language Environment for MVS & VM Programming Guide*.

---

## Chapter 19. Multitasking in PL/I

You can use PL/I multitasking to exploit multiprocessors or to synchronize and coordinate tasks in a uniprocessor. Multitasking allows data and files to be shared. This involves task creation, tasking rules, tasking program control data, attributes, options, and built-in functions.

In general, a *task* is a unit of work that competes for the resources of the computing system. A task can be independent of, or dependent on, other tasks in the system. In MVS, a task is associated with a TCB (Task Control Block) which describes the resources of the task.

In MVS you usually have one application task per address space. However, you can create multiple tasks which:

- Complete a piece of work in shorter elapsed time. You could use this in batch applications that process accumulated data.
- Use computing system resources that might be idle. This includes I/O devices as well as the CPUs.
- Effectively utilize multiprocessor complexes such as the 3090\*. However, you do not need a multiprocessing system to use multitasking.
- Implement real-time multi-user applications where the response time is critical, and each task can be processed independently as resources become available.
- Isolate independent pieces of work for reliability. This means that you can isolate the failure of a part of a job while still processing other independent parts.

---

### PL/I Multitasking Facilities

PL/I Multitasking facilities are simple extensions to the nontasking language which you can use only when running programs under MVS (excluding the IMS and CICS subsystems) on any 390 processor. The DB2 SQL statements from multiple tasks not under IMS or CICS are supported. You also must have OpenEdition Release 2 installed and activated before you can run your multitasking applications.

The run-time environment uses the POSIX multithreading services to support PL/I multitasking. Because of the differences in this environment, you might experience a different timing in your application. Also, you cannot invoke any POSIX functions while you are using the the PL/I multitasking facility. If you attempt to use the multitasking facility while you are using the POSIX-defined multithreading environment, your program will abend. See the *Language Environment for MVS & VM Debugging Guide and Run-Time Messages* for a description of message IBM05811.

For details of the run-time environment needed for using the PL/I multitasking facility, see *Language Environment for MVS & VM Programming Guide*.

---

## Creating PL/I Tasks

You must initiate multitasking in your main procedure (initial enclave). If it is found in a nested enclave, your program will be terminated and IBM0576S error message issued. See Language Environment for MVS & VM Debugging Guide and Run-Time Messages for information on this error message.

When you create (attach) a task in PL/I, you must use the CALL statement with at least one of the options TASK, EVENT, or PRIORITY. When you use one of these options, you attach a task and create a subtask. You can attach other tasks to this attached task, creating more subtasks.

Use the WAIT statement to synchronize your tasks. If the subtask synchronously invokes any procedures or functions (that is, without the TASK, EVENT, or PRIORITY options), they become part of that subtask too.

### The TASK Option of the CALL Statement

Use the TASK option of the CALL statement to specify a TASK variable. This variable is associated with the attached task and can control the dispatching priority of the attached task. You can use the task variable to interrogate or change the dispatching priority of the current task or any other task.

#### Example

```
CALL INPUT TASK(T1);
```

This call attaches a task by invoking INPUT. It names the task T1. T1 is the TASK variable.

### The EVENT Option of the CALL Statement

You can use the EVENT option to specify an EVENT variable which another task can use to WAIT for the completion of the attached task.

#### Examples

The following CALL statement creates an unnamed task:

```
CALL INPUT EVENT(E4);
```

The EVENT option indicates that this unnamed task will be finished when EVENT E4 is completed.

The following CALL statement with the TASK option creates task T4:

```
CALL INPUT TASK(T4) EVENT(E5);
```

The EVENT option allows the attaching task (or another task) to wait until event E5 is completed.

Also see the last example under “The PRIORITY Option of the CALL Statement” on page 424.

## The PRIORITY Option of the CALL Statement

You can use the PRIORITY option to set the initial dispatching priority of the attached task. This priority is only relative to the attaching task's priority. The attached task can have a priority higher than, lower than, or equal to the priority of the attaching task. The actual priority can range between zero and the MVS-allowed maximum.

If you do not use the PRIORITY option, or you set it to zero, the attached task's initial dispatching priority is the same as the attaching task's.

### Examples

The following CALL statement creates an unnamed task:

```
CALL INPUT PRIORITY(2);
```

It runs at a dispatching priority of two more than that of the attaching task.

The following CALL statement with the TASK option creates task T2:

```
CALL INPUT TASK(T2) PRIORITY(-1);
```

The PRIORITY option makes the actual dispatching priority of task T2 to be one less than that of the attaching task, but not less than zero.

The following CALL statement with the TASK option creates task T3:

```
CALL INPUT TASK(T3) PRIORITY(1) EVENT(E3);
```

The PRIORITY option makes the actual dispatching priority of task T3 to be one more than that of the attaching task, but not more than the MVS-allowed maximum. The EVENT option says the status and completion values of event E3 are set when task T3 terminates. This allows synchronization of other tasks with task T3.

---

## Synchronization and Coordination of Tasks

You can synchronize and coordinate the execution of tasks by using the EVENT option. The completion and status values control the EVENT option. The BIT(1) completion value can be '1'B which indicates the EVENT is complete or '0'B which indicates that the EVENT is not complete. The FIXED BIN(15,0) status value can be zero which shows that the EVENT has normal completion or any other number which shows that the EVENT has abnormal completion.

These values can be set by:

- The termination of a task (END, RETURN, EXIT)
- The COMPLETION pseudovalue for the completion value
- The STATUS pseudovalue for the status value
- The assignment of both EVENT variable values simultaneously
- A Statement with the EVENT option.

In addition to these conditions, you can also change the values by using a WAIT statement for an I/O or DISPLAY statement event or a CLOSE statement for a FILE having an active EVENT I/O operation.

However, while you are using these tools, you must be careful not to reset or reuse an EVENT variable already associated with an active task or I/O operation.

If you want to test the value of the EVENT variable, you can use the BUILTIN functions for the completion and status values. You can also reassign the EVENT variables to test both values (completion and status) simultaneously. You can also test using the WAIT statement, which waits for the completion of the event.

---

## Sharing Data between Tasks

If you want data to be shared between tasks you must be sure that the STATIC and current AUTOMATIC variables are known in the attaching block at the time the subtask is attached.

You must use only the latest generation of CONTROLLED variables. Subsequent generations will be known only to the allocating task (attaching or attached). Be sure that a task does not free a CONTROLLED variable shared by more than one task.

Your BASED allocations can be in AREAs shared between multiple tasks but all other BASED allocations are task local. Also, any variable of any storage class can be shared via a BASED reference as long it is allocated and is not freed until none of the tasks are using it.

When you update shared data, be sure that multiple concurrent updates are not lost. In order to do this, you must use EVENTS to coordinate updating.

---

## Sharing Files between Tasks

You can share any file subject to the following rules:

1. An attached task can share any file opened by the attaching task before the attached task is called.
2. The file must not be closed by the attaching task while it is being used by any attached task.

If the file is closed by the attaching task, results are unpredictable if an attempt is made to reference the file in any attached task, which had previously shared the file.

3. The file must not be closed by a task that did not open it.
4. Sharing must be coordinated between the sharing tasks. Most access methods do not allow simultaneous use of a file by multiple tasks. Therefore you must provide some form of interlocking. You can use WAIT and COMPLETION for interlocking in the PL/I environment.

In order to avoid the error condition associated with concurrent access, interlocking is required for files with a limit of one I/O operation. VSAM files and both SEQUENTIAL and DIRECT UNBUFFERED files not using the ISAM compatibility interface, have a limitation of one I/O operation.

To avoid competition between tasks using SYSPRINT, PL/I provides serialization for STREAM PUT statements for the STREAM file SYSPRINT only.

5. Use the EXCLUSIVE attribute to guarantee multiple updates of the same record for DIRECT UPDATE files.

You cannot share files if they are opened by the attaching task after an attached task is called. Also, attached task files are not shared by the attaching task. If you use several files to access one data set, see “Associating Several Files with One Data Set” on page 101.

---

## Producing More Reliable Tasking Programs

You will be able to produce more reliable tasking programs if you take note of the following:

- Use PROC OPTIONS(REENTRANT) on all external procedures.
- Do not modify STATIC (INTERNAL and EXTERNAL) storage.
- If the MSGFILE(SYSPRINT) run-time option is specified, the standard SYSPRINT file must be opened before any subtasks are created. Output lines from STREAM PUT statements might be interwoven with run-time messages.
- Be sure to avoid task interlocks, such as Task A waiting for Task B and Task B waiting for Task A. For more information on this, see the *PL/I for MVS & VM Language Reference*
- If you use COBOL ILC, only one task at a time can use it.
- You might not need to explicitly include certain Library routines in the PL/I Main load module if a FETCHed procedure is attached as a task (see Language Environment for MVS & VM Programming Guide for detailed information).

---

## Terminating PL/I Tasks

Termination is normal when the initial procedure of the task reaches an END or a RETURN statement. Abnormal terminations can occur:

- When control in the attached task reaches an EXIT statement.
- When control in any task reaches a STOP statement.
- When the attaching task or attaching block (calling PROCEDURE or BEGIN) terminates.
- As a result of implicit action or action on normal return for ERROR.

When a task terminates normally or abnormally, PL/I will:

- Set all incomplete I/O EVENTS to abnormal status so the results of I/O operations are undefined.
- Close all files opened by the task, disabling all I/O conditions so no I/O conditions are raised.
- Free all CONTROLLED allocations and BASED allocations (except in AREAs owned by other tasks) done by the task.
- Terminate all active blocks and subtasks.
- Post the EVENT variable on the attaching CALL statement complete.
- Unlock all records locked by the task.

---

## Dispatching Priority of Tasks

You can determine the dispatching priority of tasks in two ways while MVS can determine them in several ways. You can use the PRIORITY option to set the initial dispatching priority, as shown in the above examples. Or you can explicitly declare the actual priority as in:

```
DCL T1 TASK
```

In MVS, a task can change its own or any other task's priority. A TASK's priority value can be set or changed by:

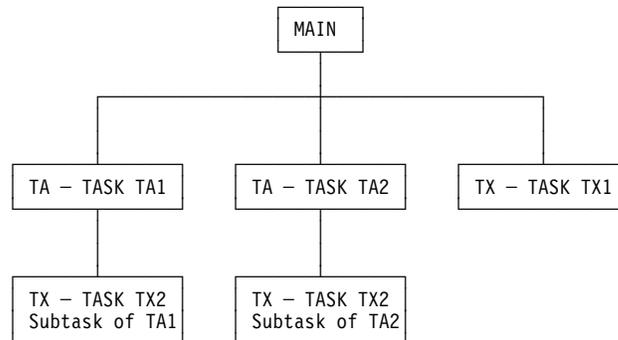
- The PRIORITY option of CALL
- The PRIORITY pseudovisible
- TASK variable assignment to an inactive TASK variable.

You can test a TASK's priority value by using the PRIORITY built-in function. However, the PRIORITY built-in function might not return the same value as was set by the PRIORITY pseudovisible. For more information, see the *PL/I for MVS & VM Language Reference*

A task hierarchy example is shown in the following program:

```
MAIN: PROC OPTIONS(MAIN REENTRANT);  
    CALL TA TASK(TA1);  
    CALL TA TASK(TA2);  
    CALL TX TASK(TX1);  
    TA: PROC;  
        CALL TX TASK(TX2);  
    END;  
END;
```

where the hierarchy is:



---

## Running Tasking Programs

When you run your tasking programs, first compile external procedures and link edit the object code, being sure to place SIBMTASK ahead of SCEELKED in SYSLIB DD statement. This rule applies to the main load module, but not to a FETCHed subroutine load module.

**Note:** Linking the main load module of an application that does not use the multitasking facility, with SIBMTASK or PLITASK, causes a loss in performance during initialization and termination.

Language Environment provides HEAP, THREADHEAP, NONIPSTACK, and PLITASKCOUNT run-time options to support PL/I multitasking. For a description of these options, see *Language Environment for MVS & VM Programming Guide*. You can also use PLIDUMP's multitasking options which are described in Chapter 17, "Using PLIDUMP" on page 386.

---

## Sample Program 1: Multiple Independent Processes

Figure 114 on page 429 is a nontasking version of a program that processes an input file consisting of transactions that can be processed independent of one another. Following that is Figure 115 on page 430, which is a tasking version of the same program.

## Multiple Independent Processes: Nontasking Version

---

```
/* PROC - MULTIPLE INDEPENDENT PROCESSES (NONTASKING VERSION) */
PROC: PROC OPTIONS(MAIN REENTRANT) REORDER;
  DCL INPUT FILE;
  DCL RECORD CHAR(80),
    REQUEST CHAR(8) DEF(RECORD);
  DCL EOF BIT INIT('0'B);
  ON ENDFILE(INPUT) EOF='1'B;
  READ FILE(INPUT) INTO(RECORD);
  DO WHILE(~EOF);
    SELECT(REQUEST);
    WHEN('REPORT') CALL REPORT; /* PRODUCE A REPORT */
    WHEN('COPY') CALL COPY; /* COPY RECORDS TO "OUTPUT" FILE */
    OTHERWISE CALL ERROR; /* INVALID */
  END;
  READ FILE(INPUT) INTO(RECORD);
END;

REPORT: PROC REORDER;
  /******
  /* PROCESS EACH RECORD TO PRODUCE A REPORT */
  /******
  DCL FREPORT FILE OUTPUT;
  WRITE FILE(FREPORT) FROM(RECORD);
END;

COPY: PROC REORDER;
  /******
  /* COPY RECORDS TO "OUTPUT" FILE */
  /******
  DCL FCOPY FILE OUTPUT;
  WRITE FILE(FCOPY) FROM(RECORD);
END;

ERROR: PROC REORDER;
  /******
  /* INVALID REQUEST - WRITE RECORD TO THE ERROR FILE */
  /******
  DCL FERROR FILE OUTPUT;
  WRITE FILE(FERROR) FROM(RECORD);
END;
END; /* MAIN */
```

---

Figure 114. Nontasking Version of Multiple Independent Processes

## Multiple Independent Processes: Tasking Version

```
/* PROCT - MULTIPLE INDEPENDENT PROCESSES (TASKING VERSION) */
PROCT: PROC OPTIONS(MAIN REENTRANT) REORDER;
DCL INPUT FILE INPUT;
DCL RECORD CHAR(80),
    REQUEST CHAR(8) DEF(RECORD);
DCL EOF BIT(1) INIT('0'B);
DCL (TASK_ENDED, WORK_READY, WORK_DONE)(3) EVENT;
DCL REC_PTR(3, /* A LIST OF RECORDS ... */
    10 /* FOR EACH TASK */
    PTR INIT((30) NULL()));
DCL
    REC_AREA(3, /* RECORD AREA (FOR EACH REC_PTR)*/
    10) CHAR(80); /* FOR EACH TASK */
DCL
    TASK_REC_PTR#(3) FIXED BIN INIT((3)0); /* INDEX INTO REC_PTR AND
    REC_AREA WHERE THE LAST RECORD
    WAS PLACED FOR EACH OF THE
    TASKS */
DCL REC_PTR# FIXED BIN;
DCL TASKS(3) ENTRY INIT(REPORT,COPY,ERROR);
DCL (FREPORT,FCOPY,FERROR) FILE OUTPUT;
DCL OUT_FILE(3) FILE INIT(FREPORT,FCOPY,FERROR);
DCL REC_TYPE FIXED BIN INIT(0);
DCL LIST_SEARCHED BIT(1);
/*****
/* START ALL TASKS AND LET THEM INITIALIZE */
/*****
STATUS(WORK_READY)=0; /* DO WORK - DON'T TERMINATE */
DO I=LBOUND(TASKS,1) TO HBOUND(TASKS,1);
    CALL TASKS(I)(OUT_FILE(I),WORK_READY(I),WORK_DONE(I),(I))/*
        MUST HAVE A TEMPORARY FOR "I" */
    EVENT(TASK_ENDED(I));
END;
/*****
/* PROCESS RECORDS */
/*****
ON ENDFILE(INPUT) EOF='1'B;
READ FILE(INPUT) INTO(RECORD);
I=LBOUND(REC_PTR,1);
DO WHILE(~EOF);
    I=REC_TYPE; /* JUST PROCESSED REC TYPE IF ANY*/
    SELECT(REQUEST);
        WHEN('REPORT') REC_TYPE=1;
        WHEN('COPY') REC_TYPE=2;
        OTHERWISE REC_TYPE=3;
    END;
    IF REC_TYPE~=I /* CURRENT TYPE NOT SAME & WE'RE */
        &I~0 THEN /* NOT HERE FOR FIRST TIME */
        COMPLETION(WORK_READY(I))='1'B; /* GET THAT TASK
            GOING IN CASE IT'S WAITING */
    LIST_SEARCHED='0'B;
```

Figure 115 (Part 1 of 3). Tasking Version of Multiple Independent Processes

---

```

PLACE_REC:
DO REC_PTR#=TASK_REC_PTR#(REC_TYPE)+1 REPEAT REC_PTR#+1;
  /*****
  /* IF LIST IS ALL FULL, WAIT FOR APPROPRIATE TASK TO BE
  /* READY FOR WORK. OTHERWISE PLACE RECORD JUST READ IN AN
  /* AVAILABLE SLOT ON THE APPROPRIATE LIST.
  *****/
IF REC_PTR#>HBOUND(REC_PTR,2) THEN
DO;
  REC_PTR#=LBOUND(REC_PTR,2); /* RESET LOOP COUNTER
  IF LIST_SEARCHED THEN
  DO;
  /*****
  /* ALL REC_PTR LIST IS EMPTY (FOR THIS REC TYPE).
  /* WAIT FOR APPROPRIATE TASK TO GET READY FOR WORK
  *****/
  WAIT(WORK_DONE(REC_TYPE));
  COMPLETION(WORK_DONE(REC_TYPE))='0'B;
  COMPLETION(WORK_READY(REC_TYPE))='1'B; /* GET THAT TASK
  GOING IN CASE IT'S WAITING
  END;
ELSE
  LIST_SEARCHED='1'B; /* WE'LL DO AT LEAST ONE COMPLETE
  SCAN OF LIST
END;
IF REC_PTR(REC_TYPE,REC_PTR#)=NULL() THEN
DO;
  REC_AREA(REC_TYPE,REC_PTR#)=RECORD; /* PUT RECORD IN
  RECORD AREA LIST
  REC_PTR(REC_TYPE,REC_PTR#)=ADDR(REC_AREA(REC_TYPE,REC_PTR#)
  ); /* SET PTR
  TASK_REC_PTR#(REC_TYPE)=REC_PTR#; /* REMEMBER THIS INDEX
  LEAVE PLACE_REC;
END;
END;
READ FILE(INPUT) INTO(RECORD);
END;
/*****
/* AT END OF JOB (END OF FILE), TELL ALL TASKS TO FINISH AND WAIT*
/* FOR THEM ALL TO FINISH
*****/
STATUS(WORK_READY)=4; /* FINISH REMAINING WORK & QUIT
COMPLETION(WORK_READY)='1'B;
WAIT(TASK_ENDED);

```

---

Figure 115 (Part 2 of 3). Tasking Version of Multiple Independent Processes

---

```

/*****
/* REPORT/COPY/ERROR TASKS */
*****/
REPORT: COPY: ERROR:
PROC(OUT_FILE,WORK_READY,WORK_DONE,MY_LIST) REORDER;
/*****
/* PROCESS "INPUT FILE" AND PRODUCE A REPORT FOR EVERY REQUEST */
*****/
DCL OUT_FILE FILE;
DCL (WORK_READY, WORK_DONE) EVENT;
DCL MY_LIST FIXED BIN;
DCL RECORD CHAR(80) BASED;
DCL LIST_SEARCHED BIT(1) INIT('0'B);
DCL J FIXED BIN;
/*****
/* DO INIT, OPEN FILES, ETC. */
*****/
;
DO J=1 REPEAT J+1;
/*****
/* PROCESS NEXT AVAILABLE RECORD */
*****/
IF REC_PTR(MY_LIST,J)≠NULL() THEN
DO;
/*****
/* PROCESS RECORD ... */
*****/
WRITE FILE(OUT_FILE) FROM(REC_PTR(MY_LIST,J)->RECORD);
REC_PTR(MY_LIST,J)=NULL();/* RECORD PROCESSED */
END;
IF J=HBOUND(REC_PTR,2) THEN
DO;
J=LBOUND(REC_PTR,2)-1; /* RESET LOOP */
IF LIST_SEARCHED THEN
DO;
/*****
/* ALL REC_PTR LIST IS EMPTY (FOR THIS REC TYPE). WAIT */
/* FOR MORE WORK OR REQUEST TO TERMINATE IF NOT ALREADY*/
/* ASKED TO TERMINATE */
*****/
IF STATUS(WORK_READY) = 4 THEN RETURN;
COMPLETION(WORK_DONE)='1'B;/* FINISHED WITH
WHAT I HAVE */
WAIT(WORK_READY); /* WAIT FOR MORE WORK OR
FOR REQUEST TO FINISH */
COMPLETION(WORK_READY)='0'B;
LIST_SEARCHED='0'B;
END;
ELSE
LIST_SEARCHED='1'B; /* WE'LL DO AT LEAST ONE COMPLETE
SCAN OF LIST LIST */
END;
END;
END; /* REPORT ... */
END; /* MAIN */

```

---

Figure 115 (Part 3 of 3). Tasking Version of Multiple Independent Processes

---

## Sample Program 2: Multiple Independent Computations

Figure 116 on page 433 is a nontasking version of a program that processes an input file and performs independent computations. Figure 117 on page 434 follows that. It is the tasking version of the same program.

## Multiple Independent Computations: Nontasking Version

---

```
/* COMP - INDEPENDENT COMPUTATIONS (NONTASKING VERSION) */
COMP: PROC OPTIONS(MAIN REENTRANT) REORDER;
DCL (AR1, AR2, AR3, A)(100,100,100) FLOAT BIN;
DCL (BR1, BR2, BR3, X)(100,100,100) FLOAT BIN;
DCL EOF BIT(1) INIT('0'B),
      (B, Y) FLOAT BIN;
DO WHILE(~EOF);
  /******
  /* READ FILE ... */
  /******

  /******
  /* 2 INDEPENDENT COMPUTATIONS FOLLOW */
  /******

  /******
  /* INDEPENDENT COMPUTATION NUMBER 1 */
  /******
  DO I=LBOUND(AR1,1)+3 TO HBOUND(AR1,1);
    DO J=LBOUND(AR1,2)+2 TO HBOUND(AR1,2);
      DO K=LBOUND(AR1,3)+1 TO HBOUND(AR1,3);
        AR1(I,J,K)=A(I,J,K)+B;
        AR2(I,J,K)=AR1(I,J,K)+AR3(I-3,J-2,K-1);
      END;
    END;
  END;

  /******
  /* INDEPENDENT COMPUTATION NUMBER 1 */
  /******
  DO I=LBOUND(BR1,1)+3 TO HBOUND(BR1,1);
    DO J=LBOUND(BR1,2)+2 TO HBOUND(BR1,2);
      DO K=LBOUND(BR1,3)+1 TO HBOUND(BR1,3);
        BR1(I,J,K)=X(I,J,K)+Y;
        BR2(I,J,K)=BR1(I,J,K)+BR3(I-3,J-2,K-1);
      END;
    END;
  END;
END;
```

---

Figure 116. Nontasking Version of Multiple Independent Computations

## Multiple Independent Computations: Tasking Version

---

```
/* COMPT - INDEPENDENT COMPUTATIONS (TASKING VERSION) */
COMPT: PROC OPTIONS(MAIN REentrant) REORDER;
DCL (AR1, AR2, AR3, A)(100,100,100) FLOAT BIN;
DCL (BR1, BR2, BR3, X)(100,100,100) FLOAT BIN;
DCL EOF BIT(1) INIT('0'B),
    (B,Y) FLOAT BIN;
DCL (AR_WORK_READY, AR_WORK_DONE) EVENT;
DCL (BR_WORK_READY, BR_WORK_DONE) EVENT;
DCL (AR_TASK, BR_TASK) EVENT;
STATUS(AR_WORK_READY),
STATUS(BR_WORK_READY)=0;          /* DO WORK - DON'T TERMINATE */
CALL AR EVENT(AR_TASK);          /* ATTACH PARALLEL */
CALL BR EVENT(BR_TASK);          /* TASKS */
DO WHILE(~EOF);
    /******
    /*
    /* READ FILE ...
    /*
    /******
    COMPLETION(AR_WORK_READY),
    COMPLETION(BR_WORK_READY)='1'B; /* GO DO IT */
    WAIT(AR_WORK_DONE, BR_WORK_DONE); /* WAIT FOR BOTH TASKS TO BE
                                     READY AGAIN */
    COMPLETION(AR_WORK_DONE),
    COMPLETION(BR_WORK_DONE)='0'B;
END;
/******
/* AT END OF JOB (END OF FILE) */
/******
STATUS(AR_WORK_READY),
STATUS(BR_WORK_READY)=4;          /* NO MORE WORK - JUST TERMINATE */
COMPLETION(AR_WORK_READY),
COMPLETION(BR_WORK_READY)='1'B; /* TERMINATE */
WAIT(AR_TASK, BR_TASK);          /* WAIT FOR BOTH TO TERMINATE */
```

---

Figure 117 (Part 1 of 2). Tasking Version of Multiple Independent Computations

---

```

/*****
/* INDEPENDENT TASK FOR COMPUTATION 1 */
*****/
AR: PROC REORDER;
DO WHILE('1'B); /* DO FOREVER */
WAIT(AR_WORK_READY);
COMPLETION(AR_WORK_READY)='0'B;
IF STATUS(AR_WORK_READY)=4 THEN
RETURN;
DO I=LBOUND(AR1,1)+3 TO HBOUND(AR1,1);
DO J=LBOUND(AR1,2)+2 TO HBOUND(AR1,2);
DO K=LBOUND(AR1,3)+1 TO HBOUND(AR1,3);
AR1(I,J,K)=A(I,J,K)+B;
AR2(I,J,K)=AR1(I,J,K)+AR3(I-3,J-2,K-1);
END;
END;
COMPLETION(AR_WORK_DONE)='1'B; /* READY FOR MORE WORK */
END;
END; /* AR */

/*****
/* INDEPENDENT TASK FOR COMPUTATION 2 */
*****/
BR: PROC REORDER;
DO WHILE('1'B); /* DO FOREVER */
WAIT(BR_WORK_READY);
COMPLETION(BR_WORK_READY)='0'B;
IF STATUS(BR_WORK_READY)=4 THEN
RETURN;
DO I=LBOUND(BR1,1)+3 TO HBOUND(BR1,1);
DO J=LBOUND(BR1,2)+2 TO HBOUND(BR1,2);
DO K=LBOUND(BR1,3)+1 TO HBOUND(BR1,3);
BR1(I,J,K)=X(I,J,K)+Y;
BR2(I,J,K)=BR1(I,J,K)+BR3(I-3,J-2,K-1);
END;
END;
COMPLETION(BR_WORK_DONE)='1'B; /* READY FOR MORE WORK */
END;
END; /* BR */
END; /* MAIN */

```

---

Figure 117 (Part 2 of 2). Tasking Version of Multiple Independent Computations

---

## Chapter 20. Interrupts and Attention Processing

To enable a PL/I program to recognize attention interrupts, two operations must be possible:

- You must be able to create an interrupt. This is done in different ways depending upon both the terminal you use and the operating system (such as VM or TSO).
- Your program must be prepared to respond to the interrupt. You can write an ON ATTENTION statement in your program so that the program receives control when the ATTENTION condition is raised.

**Note:** If the program has an ATTENTION ON-unit that you want invoked, you must compile the program with either of the following:

- The INTERRUPT option.
- A TEST option other than NOTEST or TEST(NONE,NOSYM).

Compiling this way causes INTERRUPT(ON) to be in effect, unless you explicitly specify INTERRUPT(OFF) in PLIXOPT.

You can find the procedure used to create an interrupt in the IBM instruction manual for the operating system and terminal that you are using. For TSO, see the information on the TERMINAL command and its INPUT subparameter in the *OS/VS2 TSO Command Language Reference*. For VM, see the information on the ATTN command and the discussion of external interrupts in the *VM/SP CMS Command Reference*.

There is a difference between the interrupt (the operating system recognized your request) and the raising of the ATTENTION condition.

An *interrupt* is your request that the operating system notify the running program. If a PL/I program was compiled with the INTERRUPT compile-time option, instructions are included that test an internal interrupt switch at discrete points in the program. The internal interrupt switch can be set if any program in the load module was compiled with the INTERRUPT compile-time option.

The internal switch is set when the operating system recognizes that an interrupt request was made. The execution of the special testing instructions (polling) raises the ATTENTION condition. If a debugging tool hook (or a CALL PLITEST) is encountered before the polling occurs, the debugging tool can be given control before the ATTENTION condition processing starts.

If *any* program in the load module was compiled with the INTERRUPT option, polling also takes place in all stream I/O statements to and from the terminal. Polling ensures that the ATTENTION condition is raised between PL/I statements, rather than within the statements.

Figure 118 shows a skeleton program, an ATTENTION ON-unit, and several situations where polling instructions will be generated. In the program polling will occur at:

- LABEL1
- Each iteration of the DO
- The ELSE PUT SKIP ... statement
- Block END statements

---

```
%PROCESS INTERRUPT;
.
.
.
ON ATTENTION
BEGIN;
  DCL X FIXED BINARY(15);
  PUT SKIP LIST ('Enter 1 to terminate, 0 to continue. ');
  GET SKIP LIST (X);
  IF X = 1 THEN
    STOP;
  ELSE
    PUT SKIP LIST ('Attention was ignored');
END;
.
.
.
LABEL1:
IF EMPNO ...
.
.
.
DO I = 1 TO 10;
.
.
.
END;
.
.
.
```

---

Figure 118. Using an ATTENTION ON-Unit

## Using ATTENTION ON-Units

You can use processing within the ATTENTION ON-unit to terminate potentially endless looping in a program.

Control is given to an ATTENTION ON-unit when polling instructions recognize that an interrupt has occurred. Normal return from the ON-unit is to the statement following the polling code.

## Interaction with a Debugging Tool

If the program has the TEST(ALL) or TEST(ERROR) run-time option in effect, then an interrupt causes the debugging tool to receive control the next time a hook is encountered. This might be before the program's polling code recognizes that the interrupt occurred.

Later, when the ATTENTION condition is raised, the debugging tool receives control again for condition processing.

---

## Chapter 21. Using the Checkpoint/Restart Facility

This chapter describes the PL/I Checkpoint/Restart feature which provides a convenient method of taking checkpoints during the execution of a long-running program in a batch environment.

**Note:** You cannot use Checkpoint/Restart in a TSO or VM environment.

At points specified in the program, information about the current status of the program is written as a record on a data set. If the program terminates due to a system failure, you can use this information to restart the program close to the point where the failure occurred, avoiding the need to rerun the program completely.

This restart can be either automatic or deferred. An automatic restart is one that takes place immediately (provided the operator authorizes it when requested by a system message). A deferred restart is one that is performed later as a new job.

You can request an automatic restart from within your program without a system failure having occurred.

PL/I Checkpoint/Restart uses the Advanced Checkpoint/Restart Facility of the operating system. This facility is described in the books listed in "Bibliography" on page 500.

To use checkpoint/restart you must do the following:

- Request, at suitable points in your program, that a checkpoint record is written. This is done with the built-in subroutine PLICKPT.
- Provide a data set on which the checkpoint record can be written.
- Also, to ensure the desired restart activity, you might need to specify the RD parameter in the EXEC or JOB statement (see the publication *MVS/ESA JCL Reference*).

**Note:** You should be aware of the restrictions affecting data sets used by your program. These are detailed in the "Bibliography" on page 500.

---

## Requesting a Checkpoint Record

Each time you want a checkpoint record to be written, you must invoke, from your PL/I program, the built-in subroutine PLICKPT.

```
▶—CALL—PLICKPT—(—ddname—[,—check-id—[,—org—[,—code—])—▶
```

The four arguments are all optional. If you do not use an argument, you need not specify it unless you specify another argument that follows it in the given order. In this case, you must specify the unused argument as a null string (' '). The following paragraphs describe the arguments.

### **ddname**

is a character string constant or variable specifying the name of the DD statement defining the data set that is to be used for checkpoint records. If you omit this argument, the system will use the default *ddname* SYSCHK.

### **check-id**

is a character string constant or variable specifying the name that you want to assign to the checkpoint record so that you can identify it later. If you omit this argument, the system will supply a unique identification and print it at the operator's console.

### **org**

is a character string constant or variable with the attributes CHARACTER(2) whose value indicates, in operating system terms, the organization of the checkpoint data set. PS indicates sequential (that is, CONSECUTIVE) organization; PO represents partitioned organization. If you omit this argument, PS is assumed.

### **code**

is a variable with the attributes FIXED BINARY (31), which can receive a return code from PLICKPT. The return code has the following values:

- 0 A checkpoint has been successfully taken.
- 4 A restart has been successfully made.
- 8 A checkpoint has not been taken. The PLICKPT statement should be checked.
- 12 A checkpoint has not been taken. Check for a missing DD statement, a hardware error, or insufficient space in the data set. A checkpoint will fail if taken while a DISPLAY statement with the REPLY option is still incomplete.
- 16 A checkpoint has been taken, but ENQ macro calls are outstanding and will not be restored on restart. This situation will not normally arise for a PL/I program.

## Defining the Checkpoint Data Set

You must include a DD statement in the job control procedure to define the data set in which the checkpoint records are to be placed. This data set can have either CONSECUTIVE or partitioned organization. You can use any valid ddname. If you use the ddname SYSCHK, you do not need to specify the ddname when invoking PLICKPT.

You must specify a data set name only if you want to keep the data set for a deferred restart. The I/O device can be any magnetic-tape or direct-access device.

To obtain only the last checkpoint record, then specify status as NEW (or OLD if the data set already exists). This will cause each checkpoint record to overwrite the previous one.

To retain more than one checkpoint record, specify status as MOD. This will cause each checkpoint record to be added after the previous one.

If the checkpoint data set is a library, "check-id" is used as the member-name. Thus a checkpoint will delete any previously taken checkpoint with the same name.

For direct-access storage, you should allocate enough primary space to store as many checkpoint records as you will retain. You can specify an incremental space allocation, but it will not be used. A checkpoint record is approximately 5000 bytes longer than the area of main storage allocated to the step.

No DCB information is required, but you can include any of the following, where applicable:

OPTCD=W, OPTCD=C, RECFM=UT, NCP=2, TRTCH=C

These subparameters are described in the *MVS/ESA JCL User's Guide*.

---

## Requesting a Restart

A restart can be automatic or deferred. You can make automatic restarts after a system failure or from within the program itself. The system operator must authorize all automatic restarts when requested by the system.

### Automatic Restart after a System Failure

If a system failure occurs after a checkpoint has been taken, the automatic restart will occur at the last checkpoint if you have specified RD=R (or omitted the RD parameter) in the EXEC or JOB statement.

If a system failure occurs before any checkpoint has been taken, an automatic restart, from the beginning of the job step, can still occur if you have specified RD=R in the EXEC or JOB statement.

After a system failure occurs, you can still force automatic restart from the beginning of the job step by specifying RD=RNC in the EXEC or JOB statement. By specifying RD=RNC, you are requesting an automatic step restart without checkpoint processing if another system failure occurs.

## Automatic Restart within a Program

You can request a restart at any point in your program. The rules for the restart are the same as for a restart after a system failure. To request the restart, you must execute the statement:

```
CALL PLIREST;
```

To effect the restart, the compiler terminates the program abnormally, with a system completion code of 4092. Therefore, to use this facility, the system completion code 4092 must not have been deleted from the table of eligible codes at system generation.

## Getting a Deferred Restart

To ensure that automatic restart activity is canceled, but that the checkpoints are still available for a deferred restart, specify RD=NR in the EXEC or JOB statement when the program is first executed.

▶—RESTART—(—*stepname*—, *check-id*)—▶

If you subsequently require a deferred restart, you must submit the program as a new job, with the RESTART parameter in the JOB statement. Use the RESTART parameter to specify the job step at which the restart is to be made and, if you want to restart at a checkpoint, the name of the checkpoint record.

For a restart from a checkpoint, you must also provide a DD statement that defines the data set containing the checkpoint record. The DD statement must be named SYSCHK. The DD statement must occur immediately before the EXEC statement for the job step.

## Modifying Checkpoint/Restart Activity

You can cancel automatic restart activity from any checkpoints taken in your program by executing the statement:

```
CALL PLICANC;
```

However, if you specified RD=R or RD=RNC in the JOB or EXEC statement, automatic restart can still take place from the beginning of the job step.

Also, any checkpoints already taken are still available for a deferred restart.

You can cancel any automatic restart and the taking of checkpoints, even if they were requested in your program, by specifying RD=NC in the JOB or EXEC statement.



---

## Part 7. Appendix

Appendix. Sample Program IBMLS01 .....	444
--	-----

## Appendix. Sample Program IBMLSO1

This appendix is a PL/I program that illustrates all the components of the listings produced by the compiler and the linkage editor. You can use this sample program to verify that PL/I has been installed correctly on your system.

The listings themselves are described in the chapters on compiling.

The program has comments to document both the preprocessor input and the source listing. These comments are the lines of text preceded by `/*` and followed by `*/`. Note that the `/*` does not appear in columns 1 and 2 of the input record, because `/*` in those columns is understood as a job control end-of-file statement.

In addition to the `/*` comments lines, most pages of the listing contain brief notes explaining the contents of the pages.

5688-235 IBM PL/I for MVS & VM Ver 1 Rel 1 Mod 1  
OPTIONS SPECIFIED

29 JAN 95 10:03:24 PAGE 1

```
/*PROCESS OPTIONS INSOURCE SOURCE NEST MACRO MAP STORAGE;  
/*PROCESS AGGREGATE, ESD, OFFSET;  
/*PROCESS LIST(40,45) FLAG(I) MARGINS(2,72,1) MARGINI('|');  
/*PROCESS OPT(2) TEST(ALL,SYM) ATTRIBUTES(FULL) XREF(SHORT);
```

**1**

```
00010000  
00015000  
00020000  
00030000
```

OPTIONS USED

```
AGGREGATE      NODECK      ATTRIBUTES(SHORT)  
ESD            NOGOSTMT   NOCOMPILE(S)  
GONUMBER      NOGRAPHIC  CMPAT(V2)  
INSOURCE      NOIMPRECISE FLAG(I)  
LMESSAGE      NOINCLUDE  LANGLVL(OS,SPROG)  
MACRO         NOINTERRUPT LINECOUNT(55)  
MAP           NOMDECK   LIST(40,45)  
NEST          NOSTMT    MARGINI('|')  
NUMBER        MARGINS(2,72,1)  
OBJECT        OPTIMIZE(TIME)  
OFFSET        SEQUENCE(73,80)  
OPTIONS       SIZE(2054856)  
SOURCE        NOSYNTAX(S)  
STORAGE       SYSTEM(CMS)  
              TEST(ALL,SYM)  
              XREF(SHORT)  
              TERMINAL(NOAGRAGATE,  
                        NOATTRIBUTES,  
                        NOESD,  
                        NOINSOURCE,  
                        NOLIST,  
                        NOMAP,  
                        NOOFFSET,  
                        NOOPTIONS,  
                        NOSOURCE,  
                        NOSTORAGE,  
                        NOXREF)
```

**2**

Start of the compiler listing

- 1** List of options specified in `/*PROCESS` statements.
- 2** List of options used, whether obtained by default or by being specified explicitly.

**2**  
LINE

```

1      /* PL/I Sample Program: Used to verify product installation 1      */|00040000
3      /*==== SAMPLE =====*/|00060000
4      /*====*/|00070000
5      /*====*/|00080000
6      /*==== This is the PL/I sample program that is intended to be    ==*/|00090000
7      /*==== used to verify the product's complete installation.        ==*/|00100000
8      /*==== It is expected to execute and to provide some output.     ==*/|00110000
9      /*==== Although "results" are created by the program it is only   ==*/|00120000
10     /*==== to verify that representative I/O services are operable -- ==*/|00130000
11     /*==== the results are verified (internally) by the program.      ==*/|00140000
12     /*====*/|00150000
13     /*==== The program is intended to read a data file and count     ==*/|00160000
14     /*==== the number of occurrences of each PL/I statement type.     ==*/|00170000
15     /*==== The results are displayed at the end of execution.         ==*/|00180000
16     /*==== If any count does not match the value that is expected     ==*/|00190000
17     /*==== a warning message is displayed.                             ==*/|00200000
18     /*====*/|00210000
19     /*==== When the program is executed this source program file will ==*/|00220000
20     /*==== be used as the input file. The filename or DDNAME is       ==*/|00230000
21     /*==== SOURCE.                                                    ==*/|00240000
22     /*====*/|00250000
23     /*==== NOTE: Compilation of this program should cause preprocessor ==*/|00260000
24     /*==== message:                                                  ==*/|00270000
25     /*====*/|00280000
26     /*==== IEL2250I I 140 The WORD_TABLE was successfully declared.    ==*/|00281000
27     /*====*/|00282000
28     /*==== Two compiler messages will be produced as well:          ==*/|00283000
29     /*====*/|00284000
30     /*==== IEL0533I I NO 'DECLARE' STATEMENT(S) FOR 'INDEX'.         ==*/|00285000
31     /*==== IEL0871I I 62 RESULT OF BUILTIN FUNCTION 'SUM' WILL BE    ==*/|00286000
32     /*==== EVALUATED USING FIXED POINT ARITHMETIC                    ==*/|00287000
33     /*==== OPERATIONS.                                               ==*/|00288000
34     /*====*/|00289000
35     /*====*/|00290000
36     /*====*/|00300000

38     |SAMPLE: PROCEDURE OPTIONS(MAIN) REORDER;|00320000

```

Source statements for the sample program as they appear in the input stream. These statements form the input data for the preprocessor. Pre-processor statements are identified by the % symbol.

**1** The first line of the input is included as part of the heading for all the pages of the preprocessor and compiler listing.

**2** Each input record is numbered sequentially.

**3** If an input record has a sequence number, this number is printed.

```

LINE
40      /*-----*/;00340000
41      /* The services of the PL/I Preprocessor will be used by this */;00350000
42      /* program. Since some of its variables are global (their use */;00360000
43      /* crosses macros) they must be defined early in the source */;00370000
44      /* program. */;00380000
45      /* Notice that these lines start with a percent sign and */;00390000
46      /* end with a semicolon. Notice also that they do not appear */;00400000
47      /* on the program SOURCE listing. */;00410000
48      /*-----*/;00420000
49      %DCL BIG_LIST CHAR;00430000
50      %BIG_LIST = '';00440000
51      %;00450000
52      %DCL SIZE_WORD_LIST FIXED;00460000
53      %SIZE_WORD_LIST = 0;00470000
54      %;00480000
55      %DCL MAX_WORD_LENGTH FIXED;00490000
56      %MAX_WORD_LENGTH = 0;00500000
57      %;00510000
58      %DCL CURRENT_POSITION FIXED;00520000
59      %CURRENT_POSITION = 0;00530000
60      %;00540000
61      %DCL FIRST_WORD_INDICES CHAR;00550000
62      %FIRST_WORD_INDICES = '';00560000
63      %DCL LAST_INDEX FIXED;00570000
64      %LAST_INDEX = 0;00580000
65      %;00590000
66      %ACTIVATE ADD_TO_LIST, END_OF_LIST;00600000
67      /*-----*/;00610000
68      /* End of the global Preprocessor variable declarations */;00620000
69      /*-----*/;00630000
70      /*-----*/;00640000
71      /* Non-Preprocessor data variables are declared here. Only the */;00650000
72      /* variables that are used in the main block (or in more than one */;00660000
73      /* of the contained blocks) are defined here. */;00670000
74      /*-----*/;00680000
75      /*-----*/;00690000
76      /*-----*/;00700000
77      /* Declare the source program input file and its accoutrements. */;00710000
78      /*-----*/;00720000
79      DECLARE SOURCE FILE RECORD;00730000
80      DECLARE RECORD CHARACTER(121) VARYING;00740000
81      DECLARE RECORD_READ BIT(1) INIT(FALSE);00750000
82      DECLARE LAST_CHAR_POSN FIXED BINARY(15);00760000
83      DECLARE DISCREPANCY_OCCURRED BIT(1) INIT(FALSE);00770000
84      /*-----*/;00780000
85      /*-----*/;00790000
86      /* Declare the left- and right-margins of the input dataset. */;00800000
87      /*-----*/;00810000
88      DECLARE LEFT_MARGIN FIXED BINARY(15) INIT('2');00820000
89      DECLARE RIGHT_MARGIN FIXED BINARY(15) INIT('72');00830000
90      /*-----*/;00840000

```

```

LINE
 91      /*-----*/ 00850000
 92      /* Declare '1'B as TRUE and '0'B as FALSE. */ 00860000
 93      /*-----*/ 00870000
 94      DECLARE TRUE      BIT(1) INIT('1'B); 00880000
 95      DECLARE FALSE    BIT(1) INIT('0'B); 00890000
 96      /*-----*/ 00900000
 97      /*-----*/ 00910000
 98      /* Declare which characters are acceptable as the first character */ 00920000
 99      /* of a word -- then declare acceptable succeeding characters. */ 00930000
100     /*-----*/ 00940000
101     DECLARE WORD_FIRST CHARACTERS CHAR(29) STATIC 00950000
102           INIT('ABCDEFGHIJKLMNOPQRSTUVWXYZ@#'); 00960000
103     DECLARE WORD_NEXT CHARACTERS CHAR(30) STATIC 00970000
104           INIT('ABCDEFGHIJKLMNOPQRSTUVWXYZ_@#'); 00980000
105     /*-----*/ 00990000
106     /*-----*/ 01000000
107     /* Declare a place to hold words extracted from program text. */ 01010000
108     /*-----*/ 01020000
109     DECLARE WORD_CHAR(31) VARYING; 01030000
110     DECLARE WORD_INDEX FIXED BINARY(15); 01040000
111     /*-----*/ 01050000
112     /*-----*/ 01060000
113     /* Declare the use of SYSPRINT and all of the builtin functions. */ 01070000
114     /*-----*/ 01080000
115     DECLARE SYSPRINT FILE STREAM, 01090000
116           PLIXOPT CHAR(100) VAR STATIC EXT INIT('MSGFILE(SYSPRINT)'); 01095000
117     DECLARE (HIGH, SUBSTR, SUM, UNSPEC, VERIFY) BUILTIN; 01100000
118     DECLARE ONCODE BUILTIN; 01110000

```

```

LINE
119     /*=====*/ 01120000
120     /* PL/I statement keywords are collected using the ADD_TO_LIST */ 01130000
121     /* macro. They are put into a table, WORD_TABLE, by the */ 01140000
122     /* END_OF_LIST macro. That macro also creates an index, */ 01150000
123     /* WORD_TABLE_INDEX, into the WORD_TABLE. */ 01160000
124     /* */ 01170000
125     /* Finally, a table, WORD_COUNT, is created that has a counter */ 01180000
126     /* that corresponds to each word. Whenever that word is */ 01190000
127     /* encountered in the input stream the appropriate WORD_COUNT */ 01200000
128     /* element is incremented. */ 01210000
129     /* */ 01220000
130     /* Notice that there are no semicolons in the macro statements. */ 01230000
131     /*=====*/ 01240000
132     ADD_TO_LIST ('ALLOCATE,BEGIN') 01260000
133     ADD_TO_LIST ('CALL,CLOSE,DCL,DECLARE,DEFAULT,DISPLAY') 01270000
134     ADD_TO_LIST ('DO') 01280000
135     ADD_TO_LIST ('ELSE,END,ENTRY,FREE,GENERIC,GET,GO,GOTO,IF') 01290000
136     ADD_TO_LIST ('LEAVE,LIST,LOCATE,ON,OPEN') 01300000
137     ADD_TO_LIST ('PROC,PROCEDURE') 01310000
138     ADD_TO_LIST ('READ,RETURN,REVERT,REWRITE,SELECT,SIGNAL') 01320000
139     ADD_TO_LIST ('STOP,THEN,WAIT,WHEN,WRITE') 01330000
140     END_OF_LIST 01340000
141     /*-----*/ 01350000
142     /*-----*/ 01360000
143     /* This is the table containing the results when THIS program */ 01370000
144     /* is the input dataset. There is an intentional error on the */ 01380000
145     /* IF count so that an error message can be produced. */ 01390000
146     /*-----*/ 01400000
147     DECLARE CONTROLLED_SET(SIZE_WORD_LIST) FIXED BINARY(15) 01410000
148           INIT( 0, 3, 01420000
149                 0, 1, 13, 24, 0, 2, 01430000
150                 14, 01440000
151                 13, 23, 0, 0, 0, 0, 1, 0, 14, 01450000
152                 0, 7, 0, 4, 1, 01460000
153                 2, 3, 01470000
154                 2, 4, 0, 0, 1, 0, 01480000
155                 2, 13, 0, 2, 0); 01490000

```

```

LINE
157      /*=====*/ 01510000
158      /*= SAMPLE will perform the following tasks:          =*/ 01520000
159      /*= 1) OPEN the input dataset                          =*/ 01530000
160      /*= 2) READ each record and, for each record,         =*/ 01540000
161      /*= a) Extract a character string that meets the PL/I  =*/ 01550000
162      /*=      definition of a word.                          =*/ 01560000
163      /*= b) If the word also appears in the list of interesting =*/ 01570000
164      /*=      words, record its presence by incrementing a counter. =*/ 01580000
165      /*= 3) Report on the number of appearances of the words that =*/ 01590000
166      /*=      actually appeared in the dataset.             =*/ 01600000
167      /*= 4) DISPLAY a message if the count does not match the count =*/ 01610000
168      /*=      of PL/I statement keywords in this program.    =*/ 01620000
169      /*=====*/ 01630000
170      01640000
171      /*-----*/ 01650000
172      /* Describe the action to take on selected exceptional conditions. */ 01660000
173      /*-----*/ 01670000
174      01680000
175      /*-----*/ 01690000
176      /* If the file has not been properly defined, tell them about it. */ 01700000
177      /*-----*/ 01710000
178      ON UNDEFINEDFILE(SOURCE) 01720000
179      BEGIN; 01730000
180      DISPLAY ('The input data set has not been defined. '); 01740000
181      STOP; 01750000
182      END; 01760000
183      01770000
184      /*-----*/ 01780000
185      /* When the file has been processed indicate "no record read" */ 01790000
186      /*-----*/ 01800000
187      ON ENDFILE(SOURCE) 01810000
188      BEGIN; 01820000
189      RECORD_READ = FALSE; 01830000
190      END; 01840000
191      01850000
192      /*-----*/ 01860000
193      /* If any other errors occur, write a message and terminate. */ 01870000
194      /*-----*/ 01880000
195      ON ERROR 01890000
196      BEGIN; 01900000
197      ON ERROR SYSTEM; 01910000
198      DISPLAY ('Unspecified error occurred. ONCODE=' || ONCODE ); 01920000
199      STOP; 01930000
200      END; 01940000
201      01950000
202      /*-----*/ 01960000
203      /* Prepare the input dataset for processing -- mark it as open. */ 01970000
204      /*-----*/ 01980000
205      OPEN FILE(SOURCE) INPUT; 01990000

```

```

LINE
206      /*=====*/ 02000000
207      /* Count the use of PL/I statements in each record of the */ 02010000
208      /* input data set. */ 02020000
209      /*=====*/ 02030000
210      02040000
211      /*-----*/ 02050000
212      /* Read the first record of the input dataset. */ 02060000
213      /*-----*/ 02070000
214      RECORD_READ = TRUE;
215      READ FILE(SOURCE) INTO (RECORD);
216      02090000
217      02100000
218      /*-----*/ 02110000
219      /* Process the first and all succeeding records. */ 02120000
220      /*-----*/ 02130000
221      DO WHILE (RECORD_READ);
222      02140000
223      /* Set the "last character" position to the left margin */ 02150000
224      LAST_CHAR POSN = LEFT_MARGIN;
225      /* Use NEXT_WORD to extract the first word from this record. */ 02160000
226      WORD = NEXT_WORD(RECORD);
227      02170000
228      /*-----*/ 02180000
229      /* Extract words from this record until no more remain. */ 02190000
230      /*-----*/ 02200000
231      DO WHILE (WORD ^= '');
232      /* Use LOOKUP_WORD to find its position in the table. */ 02210000
233      WORD_INDEX = LOOKUP_WORD(WORD);
234      02220000
235      /*-----*/ 02230000
236      /* If the word is in the list, count it. */ 02240000
237      /*-----*/ 02250000
238      IF WORD_INDEX ^= 0 THEN
239      WORD_COUNT(WORD_INDEX) = WORD_COUNT(WORD_INDEX) + 1;
240      ELSE;
241      /* Get the next word from the record. */ 02260000
242      WORD = NEXT_WORD(RECORD);
243      02270000
244      END;
245      02280000
246      /*-----*/ 02290000
247      /* Read the next record from the input data set. */ 02300000
248      /*-----*/ 02310000
249      READ FILE(SOURCE) INTO (RECORD);
250      02320000
251      END;
252      02330000
253      /*-----*/ 02340000
254      /* Input from the data set is exhausted. CLOSE it. */ 02350000
255      /*-----*/ 02360000
256      CLOSE FILE(SOURCE);
257      02370000

```

```

LINE
255      /*=====*/ 02490000
256      /*= The report that details and summarizes the use of word in the =*/ 02500000
257      /*= WORD_TABLE is prepared in this section. =*/ 02510000
258      /*=====*/ 02520000
259
260      PUT SKIP LIST (' ***** '); 02530000
261      PUT SKIP LIST (' *** Word-use Report *** '); 02540000
262      PUT SKIP LIST (' ***** '); 02550000
263      PUT SKIP LIST (' -count- --word-- '); 02560000
264
265      /*-----*/ 02570000
266      /* Review the activity for each word in the list. */ 02580000
267      /*-----*/ 02590000
268      DO WORD_INDEX = 1 TO SIZE_WORD_LIST; 02600000
269
270      /*-----*/ 02610000
271      /* If the word was used then display the word and its use-count. */ 02620000
272      /*-----*/ 02630000
273      IF WORD_COUNT(WORD_INDEX) > 0 THEN 02640000
274          PUT SKIP EDIT (WORD_COUNT(WORD_INDEX), 02650000
275                        WORD_TABLE(WORD_INDEX)) 02660000
276                        (F(6), X(6), A); 02670000
277      ELSE; 02680000
278
279      /*-----*/ 02690000
280      /* If there was a discrepancy between what was counted and what */ 02700000
281      /* was expected then display a warning message and remember that */ 02710000
282      /* it had occurred. */ 02720000
283      /*-----*/ 02730000
284      IF WORD_COUNT(WORD_INDEX) /= CONTROLLED_SET(WORD_INDEX) THEN 02740000
285          DO; 02750000
286              PUT SKIP EDIT ((12)'-', 02760000
287                            'The previous value should have been', 02770000
288                            CONTROLLED_SET(WORD_INDEX)) 02780000
289                            (A, A, F(6)); 02790000
290              DISCREPANCY_OCCURRED = TRUE; 02800000
291          END; 02810000
292      ELSE; 02820000
293      END; 02830000
294
295      /*-----*/ 02840000
296      /* Summarize word activity on this input dataset. */ 02850000
297      /*=====*/ 02860000
298
299      PUT SKIP(2) LIST ('There were ' || SUM(WORD_COUNT) 02870000
300                      || ' references to ' || SIZE_WORD_LIST 02880000
301                      || ' words. '); 02890000
302
303      /*-----*/ 02900000
304      /* If a discrepancy between one of the counts and the expected */ 02910000
305      /* counts occurred then display a warning message. */ 02920000
306      /*-----*/ 02930000
307      IF DISCREPANCY_OCCURRED THEN 02940000
308          PUT SKIP(2) LIST ('There was a discrepancy in at least one of' 02950000
309                          || ' the word-counts. '); 02960000
310      ELSE; 02970000

```

```

LINE
295      /*=====*/ 02890000
296      /* Summarize word activity on this input dataset. */ 02900000
297      /*=====*/ 02910000
298
299      PUT SKIP(2) LIST ('There were ' || SUM(WORD_COUNT) 02920000
300                      || ' references to ' || SIZE_WORD_LIST 02930000
301                      || ' words. '); 02940000
302
303      /*-----*/ 02950000
304      /* If a discrepancy between one of the counts and the expected */ 02960000
305      /* counts occurred then display a warning message. */ 02970000
306      /*-----*/ 02980000
307      IF DISCREPANCY_OCCURRED THEN 02990000
308          PUT SKIP(2) LIST ('There was a discrepancy in at least one of' 03000000
309                          || ' the word-counts. '); 03010000
310      ELSE; 03020000

```

```

LINE
311      /*==== NEXT_WORD =====*/ 03050000
312      /*=====*/ 03060000
313      /*== ==*/ 03070000
314      /*== Extract a word from the argument string that is passed. ==*/ 03080000
315      /*== Return it as CHAR(31) VARYING. ==*/ 03090000
316      /*== ==*/ 03100000
317      /*== Ignore PL/I comments and constants (strings surrounded by ==*/ 03110000
318      /*== single quotes ('). Comments and constants can not be ==*/ 03120000
319      /*== continued but must be complete in the argument string. ==*/ 03130000
320      /*== ==*/ 03140000
321      /*== If no more words exist then a null character string will ==*/ 03150000
322      /*== be returned. ==*/ 03160000
323      /*== ==*/ 03170000
324      /*=====*/ 03180000
325      /*=====*/ 03190000
326      03200000
327      NEXT_WORD: PROCEDURE(DATA_RECORD) RETURNS(CHAR(31) VARYING); 03210000
328      03220000
329      DECLARE DATA_RECORD      CHAR(*) VARYING; 03230000
330      DECLARE DATA_WORD        CHAR(31) VARYING; 03240000
331      DECLARE NEXT_CHARACTER     CHAR(1); 03250000
332      DECLARE LENGTH_OF_STRING  FIXED BINARY(15); 03260000
333      03270000
334      DECLARE NEXT_CHAR_POSN     FIXED BINARY(15); 03280000
335      03290000
336      /*=====*/ 03300000
337      /*= LAST_CHAR_POSN remembers, from call to call, the point where==*/ 03310000
338      /*= the search for additional words will start. Management of ==*/ 03320000
339      /*= its value is a key concern to this function. ==*/ 03330000
340      /*= ==*/ 03340000
341      /*= Comments and constants in the argument string will be ==*/ 03350000
342      /*= ignored. If a character is found that is a legitimate PL/I ==*/ 03360000
343      /*= "first-character" then a word is assumed to follow. It ==*/ 03370000
344      /*= will be built by concatenating (suffixing) additional, ==*/ 03380000
345      /*= legitimate "next-characters." ==*/ 03390000
346      /*=====*/ 03400000

```

```

LINE
347      /*=====*/ 03410000
348      /*= Scan each character in the record. Start at the position ==*/ 03420000
349      /*= where scanning last terminated (LAST_CHAR_POSN) and ==*/ 03430000
350      /*= continue until the end of a word or the end of the record ==*/ 03440000
351      /*= is reached. ==*/ 03450000
352      /*=====*/ 03460000
353      03470000
354      DATA_WORD = ''; 03480000
355      DO NEXT_CHAR_POSN = LAST_CHAR_POSN TO RIGHT_MARGIN 03490000
356      WHILE (DATA_WORD = ''); 03500000
357      NEXT_CHARACTER = SUBSTR(DATA_RECORD, NEXT_CHAR_POSN, 1); 03510000
358      SELECT (NEXT_CHARACTER); 03520000
359      03530000
360      WHEN ('/') 03540000
361      03550000
362      /*-----*/ 03560000
363      /*= If this turns out to be a comment then skip over it. ==*/ 03570000
364      /*-----*/ 03580000
365      DO; 03590000
366      IF SUBSTR(DATA_RECORD, NEXT_CHAR_POSN, 2) = '/' THEN 03600000
367      NEXT_CHAR_POSN = NEXT_CHAR_POSN + 3 03610000
368      + INDEX(SUBSTR(DATA_RECORD, NEXT_CHAR_POSN+2), '* /'); 03620000
369      ELSE; 03630000
370      END; 03640000
371      03650000
372      WHEN ('''') 03660000
373      03670000
374      /*-----*/ 03680000
375      /*= Skip over the constant. ==*/ 03690000
376      /*-----*/ 03700000
377      NEXT_CHAR_POSN = NEXT_CHAR_POSN 03710000
378      + INDEX(SUBSTR(DATA_RECORD, NEXT_CHAR_POSN+1), '''); 03720000

```

```

LINE
379      /*=====*/ 03730000
380      /*= This may be the start of a word.  Extract it if so.  =*/ 03740000
381      /*=====*/ 03750000
382      OTHERWISE 03760000
383      03770000
384      03780000
385      /*-----*/ 03790000
386      /* This may be the start of a word.  */ 03800000
387      /*-----*/ 03810000
388      DO; 03820000
389      03830000
390      /*-----*/ 03840000
391      /* If the next character is not acceptable as the first  */ 03850000
392      /* character of a word then do nothing further -- our  */ 03860000
393      /* enclosing DO will step to the next character for  */ 03870000
394      /* further checking.  */ 03880000
395      /*-----*/ 03890000
396      IF INDEX(WORD_FIRST_CHARACTERS, NEXT_CHARACTER) = 0 THEN; 03900000
397      ELSE 03910000
398      03920000
399      /*-----*/ 03930000
400      /* This is the start of a word.  Collect the rest of it*/ 03940000
401      /*-----*/ 03950000
402      03960000
403      DO; 03970000
404      DATA_WORD = NEXT_CHARACTER; 03980000
405      03990000
406      /*-----*/ 04000000
407      /* Build up DATA_WORD by iteratively appending  */ 04010000
408      /* characters from the input argument string.  Do it */ 04020000
409      /* as long as the characters are acceptable PL/I  */ 04030000
410      /* "next=characters."  */ 04040000
411      /*-----*/ 04050000
412      04060000
413      DO NEXT_CHAR_POSN = NEXT_CHAR_POSN+1 TO RIGHT_MARGIN 04070000
414      WHILE (INDEX(WORD_NEXT_CHARACTERS, 04080000
415      SUBSTR(DATA_RECORD, NEXT_CHAR_POSN,1)) ^= 0); 04090000
416      DATA_WORD = DATA_WORD 04100000
417      || SUBSTR(DATA_RECORD, NEXT_CHAR_POSN,1); 04110000
418      END; 04120000
419      LAST_CHAR_POSN = NEXT_CHAR_POSN + 1; 04130000
420      END; 04140000
421      END; 04150000
422      END; /* End of the SELECT (NEXT_CHARACTER) statement */ 04160000
423      END; /* End of the DO that tries to find a word */ 04170000
424      04180000
425      RETURN (DATA_WORD); 04190000
426      END; 04200000

```

```

LINE
427      /*== LOOKUP_WORD =====*/ 04210000
428      /*=====*/ 04220000
429      /*==                               ==*/ 04230000
430      /*== Find the word in the WORD_TABLE that matches the ==*/ 04240000
431      /*== argument string (CHAR(*) VARYING) and return the ==*/ 04250000
432      /*== position of that word (its subscript) to the ==*/ 04260000
433      /*== invoker (FIXED BINARY(15)). ==*/ 04270000
434      /*==                               ==*/ 04280000
435      /*== If the word does not exist in the list a 0 will be ==*/ 04290000
436      /*== returned. ==*/ 04300000
437      /*==                               ==*/ 04310000
438      /*=====*/ 04320000
439      /*=====*/ 04330000
440      04340000
441      LOOKUP_WORD: PROCEDURE(DATA_WORD) RETURNS(FIXED BINARY(15)); 04350000
442      04360000
443      DECLARE DATA_WORD CHAR(*) VARYING; 04370000
444      DECLARE WORD_NUMBER FIXED BINARY(15); 04380000
445      04390000
446      /*=====*/ 04400000
447      /* A sequential search is used to locate the required word. ==*/ 04410000
448      /* WORD_INDEX_TABLE is used to start the search at the first ==*/ 04420000
449      /* word in the list that has the same first character. ==*/ 04430000
450      /*=====*/ 04440000
451      WORD_NUMBER = WORD_INDEX_TABLE /* Subscript is on next line */ 04450000
452      (INDEX WORD_FIRST_CHARACTERS, SUBSTR(DATA_WORD,1,1)); 04460000
453      04470000
454      /*-----*/ 04480000
455      /* Search words in the WORD_TABLE until the word is found or is */ 04490000
456      /* determined to be not a part of the list -- its index number */ 04500000
457      /* (WORD_NUMBER) is zero. */ 04510000
458      /*-----*/ 04520000
459      DO WORD_NUMBER = WORD_NUMBER BY 1 04530000
460      UNTIL ( WORD_TABLE(WORD_NUMBER) = DATA_WORD 04540000
461      | WORD_NUMBER = 0); 04550000
462      04560000
463      /*-----*/ 04570000
464      /* If the word in the WORD_TABLE is alphabetically greater */ 04580000
465      /* than the argument word then a match cannot be found. Set */ 04590000
466      /* the WORD_NUMBER to 0 to indicate a non=match situation. */ 04600000
467      /*-----*/ 04610000
468      IF WORD_TABLE(WORD_NUMBER) > DATA_WORD THEN 04620000
469      WORD_NUMBER = 0; 04630000
470      ELSE; 04640000
471      END; 04650000
472      04660000
473      RETURN (WORD_NUMBER); 04670000
474      END; 04680000
475      %; 04690000
476      %/***** 04700000
477      %/***** 04710000

```

```

LINE
478      /**=                               =*/; 04720000
479      /**= Submit a list of words that are to be included in the      =*/; 04730000
480      /**= WORD_TABLE. They must be in alphabetic order. The list    =*/; 04740000
481      /**= must be within parentheses and cannot contain any blanks =*/; 04750000
482      /**=                               =*/; 04760000
483      /*=====                          =*/; 04770000
484      /*=====                          =*/; 04780000
485      %ADD TO LIST: PROC(WORD_LIST) RETURNS(CHAR);                      04790000
486          DCL WORD_LIST      CHAR;                                     04800000
487          DCL EXTRACTED_WORD CHAR;                                     04810000
488          DCL THIS_INDEX_CHAR CHAR;                                   04820000
489          DCL THIS_INDEX     FIXED;                                   04830000
490          DCL COMMA          FIXED;                                   04840000
491          DCL (INDEX, LENGTH, SUBSTR) BUILTIN;                       04850000
492                                                                 04860000
493          /* Remove the leading and trailing apostrophes                */ 04870000
494          WORD_LIST = SUBSTR(WORD_LIST,2,LENGTH(WORD_LIST)-2);        04880000
495                                                                 04890000
496          /*-----*/ 04900000
497          /* Extract each word from the argument data string. Put each */ 04910000
498          /* word onto the end of a growing string that will eventually */ 04920000
499          /* be used in the INITIAL clause of the declaration of        */ 04930000
500          /* the WORD_TABLE.                                           */ 04940000
501          /*-----*/ 04950000
502      PARSE_LOOP: 04960000
503          /* Keep track of the size of the word list.                    */ 04970000
504          SIZE_WORD_LIST = SIZE_WORD_LIST + 1;                          04980000
505          THIS_INDEX = INDEX('ABCDEFGHIJKLMNPQRSTUVWXYZ',              04990000
506                          SUBSTR(WORD_LIST,1,1));                       05000000
507                                                                 05010000
508          /*-----*/ 05020000
509          /* Is this is the first time that this initial character has   */ 05030000
510          /* been encountered? If so, the table of indices into         */ 05040000
511          /* WORD_TABLE must be updated.                                 */ 05050000
512          /*-----*/ 05060000
513          IF THIS_INDEX /= LAST_INDEX THEN                               05070000
514              DO; 05080000
515                                                                 05090000
516          /*-----*/ 05100000
517          /* Update the "initial character index" for all characters     */ 05110000
518          /* between the last one and this one.                          */ 05120000
519          /*-----*/ 05130000
520                                                                 05140000
521          DO LAST_INDEX = LAST_INDEX+1 TO THIS_INDEX;                   05150000
522          /*-----*/ 05160000
523          /* If the last character processed and this character are     */ 05170000
524          /* are not alphabetically adjacent then a zero (no word      */ 05180000
525          /* having such a first character is acceptable) must be      */ 05190000
526          /* appended to the list of indices. The list will be         */ 05200000
527          /* used later in the INITIAL clause of the declaration        */ 05210000
528          /* of WORD_TABLE_INDEX.                                       */ 05220000

```

```

LINE
529      /*-----*/ 05230000
530      IF LAST_INDEX < THIS_INDEX THEN 05240000
531          FIRST_WORD_INDICES = FIRST_WORD_INDICES || ', 0'; 05250000
532      ELSE 05260000
533          FIRST_WORD_INDICES = FIRST_WORD_INDICES||','||SIZE_WORD_LIST; 05270000
534      END; 05280000
535      LAST_INDEX = LAST_INDEX - 1; 05290000
536      END; 05300000
537      ELSE; 05310000
538 05320000
539      COMMA = INDEX(WORD_LIST,','); 05330000
540 05340000
541      /*-----*/ 05350000
542      /* Is there a comma after this word? */ 05360000
543      /*-----*/ 05370000
544      IF COMMA = 0 THEN 05380000
545          DO; 05390000
546 05400000
547          /*-----*/ 05410000
548          /* Since this word is not followed by a comma it is the */ 05420000
549          /* last one in the list. */ 05430000
550          /*-----*/ 05440000
551          BIG_LIST = BIG_LIST || ' ' || WORD_LIST || ' ', '; 05450000
552 05460000
553          /*-----*/ 05470000
554          /* Keep track of the longest word in the list. */ 05480000
555          /*-----*/ 05490000
556          IF LENGTH(WORD_LIST) > MAX_WORD_LENGTH THEN 05500000
557              MAX_WORD_LENGTH = LENGTH(WORD_LIST); 05510000
558          ELSE; 05520000
559          RETURN(''); 05530000
560      END; 05540000
561      ELSE 05550000
562          DO; 05560000
563 05570000
564          /*-----*/ 05580000
565          /* Extract the next word and remove it from the input. */ 05590000
566          /*-----*/ 05600000
567          EXTRACTED_WORD = SUBSTR(WORD_LIST,1,COMMA-1); 05610000
568          BIG_LIST = BIG_LIST || ' ' || 05620000
569              EXTRACTED_WORD || ' ', '; 05630000
570 05640000
571          /*-----*/ 05650000
572          /* Keep track of the longest word in the list. */ 05660000
573          /*-----*/ 05670000
574          IF LENGTH(EXTRACTED_WORD) > MAX_WORD_LENGTH THEN 05680000
575              MAX_WORD_LENGTH = LENGTH(EXTRACTED_WORD); 05690000
576          ELSE; 05700000
577          /* Remove this word and the comma from the input string. */ 05710000
578          WORD_LIST = SUBSTR(WORD_LIST,COMMA+1); 05720000
579      END; 05730000

```

```

LINE
580      GO TO PARSE_LOOP;                                05740000
581      %END;                                           05750000
582      %;                                               05760000
583      %/*****;/                                       05770000
584      %/*****;/                                       05780000
585      %/== =*/;                                        05790000
586      %/== All words contained in the search list have been submitted. =*/; 05800000
587      %/== Create the DECLARES for the WORD_TABLE, WORD_COUNT vector =*/; 05810000
588      %/== and the WORD_INDEX_TABLE. =*/;            05820000
589      %/== =*/;                                        05830000
590      %/*****;/                                       05840000
591      %/*****;/                                       05850000
592      %END OF LIST: PROC RETURNS(CHAR);                05860000
593      DCL TABLE_DCL CHAR;                              05870000
594                                                    05880000
595      /*-----*/                                       05890000
596      /* Create the DECLARE for the WORD_TABLE */      05900000
597      /*-----*/                                       05910000
598      TABLE_DCL = 'DECLARE '                            05920000
599                || 'WORD_TABLE(' || (SIZE_WORD_LIST+1) || ') ' 05930000
600                || 'CHAR(' || MAX_WORD_LENGTH || ') '      05940000
601                || 'INIT(' || BIG_LIST                     05950000
602                || 'HIGH(' || MAX_WORD_LENGTH || '));' ;    05960000
603                                                    05970000
604      /*-----*/                                       05980000
605      /* Append the DECLARE for the WORD_COUNT array. */ 05990000
606      /*-----*/                                       06000000
607      TABLE_DCL = TABLE_DCL                            06010000
608                || 'DECLARE WORD_COUNT(' || SIZE_WORD_LIST || ') ' 06020000
609                || 'FIXED BINARY(15) INIT((' || SIZE_WORD_LIST || ')0);' ; 06030000
610                                                    06040000
611      /*-----*/                                       06050000
612      /* Append the DECLARE for the WORD_INDEX_TABLE array. */ 06060000
613      /*-----*/                                       06070000
614      TABLE_DCL = TABLE_DCL                            06080000
615                || 'DECLARE WORD_INDEX_TABLE(26) FIXED BINARY(15) INIT(' ; 06090000
616      TABLE_DCL = TABLE_DCL || SUBSTR(FIRST_WORD_INDICES,5); 06100000
617                                                    06110000
618      /*-----*/                                       06120000
619      /* If the last word started with a Z then the initial values */ 06130000
620      /* for the index table is complete. If not then some zeroes */ 06140000
621      /* have to be added to account for all 26 array items. */ 06150000
622      /*-----*/                                       06160000
623      IF SIZE_WORD_LIST = 26 THEN                          06170000
624          TABLE_DCL = TABLE_DCL || ') ' ;              06180000
625      ELSE                                                  06190000
626          TABLE_DCL = TABLE_DCL || ',(' || 26-LAST_INDEX || ')0);' ; 06200000
627      NOTE ('The WORD_TABLE was successfully declared.',0); 06205000
628      RETURN(TABLE_DCL);                                  06210000
629      %END;                                               06220000
630      END;                                                 06230000

```

PREPROCESSOR DIAGNOSTIC MESSAGES

**1** **2** **3**  
 ERROR ID L LINE MESSAGE DESCRIPTION

PREPROCESSOR INFORMATORY MESSAGES

IEL2250I I 140 The WORD\_TABLE was successfully declared.

END OF PREPROCESSOR DIAGNOSTIC MESSAGES

Diagnostic messages generated by the preprocessor. All messages generated by the compiler (including the preprocessor) are documented in the publication *PL/I MVS & VM Compile-Time Messages and Codes*.

- 1** ERROR ID identifies the message originating from the compiler (IEL) and gives the message number.
- 2** L indicates the severity level of the message.
- 3** LINE lists the number of the line in which the error occurred.

STMT LEV NT SOURCE LISTING

**4**  
R

```

/* PL/I Sample Program: Used to verify product installation */|00040000

/*=== SAMPLE =====*/|00060000
/*=====*/|00070000
/*=== */|00080000
/*== This is the PL/I sample program that is intended to be */|00090000
/*== used to verify the product's complete installation. */|00100000
/*== It is expected to execute and to provide some output. */|00110000
/*== Although "results" are created by the program it is only */|00120000
/*== to verify that representative I/O services are operable -- */|00130000
/*== the results are verified (internally) by the program. */|00140000
/*== */|00150000
/*== The program is intended to read a data file and count */|00160000
/*== the number of occurrences of each PL/I statement type. */|00170000
/*== The results are displayed at the end of execution. */|00180000
/*== If any count does not match the value that is expected */|00190000
/*== a warning message is displayed. */|00200000
/*== */|00210000
/*== When the program is executed this source program file will */|00220000
/*== be used as the input file. The filename or DDNAME is */|00230000
/*== SOURCE. */|00240000
/*== NOTE: Compilation of this program should cause preprocessor */|00250000
/*== message: */|00260000
/*== */|00270000
/*== IEL2250I I 140 The WORD_TABLE was successfully declared. */|00280000
/*== */|00281000
/*== Two compiler messages will be produced as well: */|00282000
/*== */|00283000
/*== IEL0533I I NO 'DECLARE' STATEMENT(S) FOR 'INDEX'. */|00284000
/*== IEL0871I I 62 RESULT OF BUILTIN FUNCTION 'SUM' WILL BE */|00285000
/*== EVALUATED USING FIXED POINT ARITHMETIC */|00286000
/*== OPERATIONS. */|00287000
/*== */|00288000
/*== */|00289000
/*=====*/|00290000
/*=====*/|00300000

1 0 |SAMPLE: PROCEDURE OPTIONS(MAIN) REORDER; |00320000

```

Source listing. This is the output from the preprocessor and the input to the compiler. All the preprocessor statements have been executed and all preprocessor comments have been deleted.

**4** Numbers in this column of the listing indicate the maximum depth of replacement of preprocessor statements.

```

/*=====*/ 00640000
/* Non-Preprocessor data variables are declared here. Only the */ 00650000
/* variables that are used in the main block (or in more than one */ 00660000
/* of the contained blocks) are defined here. */ 00670000
/*=====*/ 00680000
00690000
/*-----*/ 00700000
/* Declare the source program input file and its accoutrements. */ 00710000
/*-----*/ 00720000
2 1 0 DECLARE SOURCE FILE RECORD; 00730000
3 1 0 DECLARE RECORD CHARACTER(121) VARYING; 00740000
4 1 0 DECLARE RECORD_READ BIT(1) INIT(FALSE); 00750000
5 1 0 DECLARE LAST_CHAR_POSN FIXED BINARY(15); 00760000
6 1 0 DECLARE DISCREPANCY_OCCURRED BIT(1) INIT(FALSE); 00770000
00780000
/*-----*/ 00790000
/* Declare the left- and right-margins of the input dataset. */ 00800000
/*-----*/ 00810000
7 1 0 DECLARE LEFT_MARGIN FIXED BINARY(15) INIT('2'); 00820000
8 1 0 DECLARE RIGHT_MARGIN FIXED BINARY(15) INIT('72'); 00830000
00840000
/*-----*/ 00850000
/* Declare '1'B as TRUE and '0'B as FALSE. */ 00860000
/*-----*/ 00870000
9 1 0 DECLARE TRUE BIT(1) INIT('1'B); 00880000
10 1 0 DECLARE FALSE BIT(1) INIT('0'B); 00890000
00900000
/*-----*/ 00910000
/* Declare which characters are acceptable as the first character */ 00920000
/* of a word -- then declare acceptable succeeding characters. */ 00930000
/*-----*/ 00940000
11 1 0 DECLARE WORD_FIRST_CHARACTERS CHAR(29) STATIC 00950000
INIT('ABCDEFGHIJKLMNQRSTUWXYZ@#'); 00960000
12 1 0 DECLARE WORD_NEXT_CHARACTERS CHAR(30) STATIC 00970000
INIT('ABCDEFGHIJKLMNQRSTUWXYZ_@#'); 00980000
00990000
/*-----*/ 01000000
/* Declare a place to hold words extracted from program text. */ 01010000
/*-----*/ 01020000
13 1 0 DECLARE WORD_CHAR(31) VARYING; 01030000
14 1 0 DECLARE WORD_INDEX FIXED BINARY(15); 01040000
01050000
/*-----*/ 01060000
/* Declare the use of SYSPRINT and all of the builtin functions. */ 01070000
/*-----*/ 01080000
15 1 0 DECLARE SYSPRINT FILE STREAM, 01090000
PLIXOPT CHAR(100) VAR STATIC EXT INIT('MSGFILE(SYSPRINT)'); 01095000
16 1 0 DECLARE (HIGH, SUBSTR, SUM, UNSPEC, VERIFY) BUILTIN; 01100000
17 1 0 DECLARE ONCODE BUILTIN; 01110000

```

```

/*=====*/ 01120000 R
/* PL/I statement keywords are collected using the ADD_TO_LIST */ 01130000
/* macro. They are put into a table, WORD_TABLE, by the */ 01140000
/* END_OF_LIST macro. That macro also creates an index, */ 01150000
/* WORD_TABLE_INDEX, into the WORD_TABLE. */ 01160000
/* */ 01170000
/* Finally, a table, WORD_COUNT, is created that has a counter */ 01180000
/* that corresponds to each word. Whenever that word is */ 01190000
/* encountered in the input stream the appropriate WORD_COUNT */ 01200000
/* element is incremented. */ 01210000
/* */ 01220000
/* Notice that there are no semicolons in the macro statements. */ 01230000
/*=====*/ 01240000
01260000 1
01270000 1
01280000 1
01290000 1
01300000 1
01310000 1
01320000 1
01330000 1
18 1 0 DECLARE WORD_TABLE( 37) CHAR( 9)INIT('ALLOCATE', 'BEGIN', 01340000 1
'CALL', 'CLOSE', 'DCL', 'DECLARE', 'DEFAULT', 'DISPLAY', 'DO', 'ELSE', 01340000 1
'END', 'ENTRY', 'FREE', 'GENERIC', 'GET', 'GO', 'GOTO', 'IF', 'LEAVE', 01340000 1
'LIST', 'LOCATE', 'ON', 'OPEN', 'PROC', 'PROCEDURE', 'READ', 'RETURN', 01340000 1
'REVERT', 'REWRITE', 'SELECT', 'SIGNAL', 'STOP', 'THEN', 'WAIT', 'WHEN' 01340000 1
19 1 0 , 'WRITE', HIGH( 9));DECLARE WORD_COUNT( 36) FIXED BINARY(15) 01340000 1
20 1 0 ) INIT(( 36)0);DECLARE WORD_INDEX_TABLE(26) FIXED BINARY(15) INIT( 01340000 1
1, 2, 3, 5, 10, 13, 14, 0, 18 01340000 1
, 0, 0, 19, 0, 0, 22, 24, 0, 26, 30, 33, 01340000 1
0, 0, 34,( 3)0); 01340000 1
01350000
/*-----*/ 01360000
/* This is the table containing the results when THIS program */ 01370000
/* is the input dataset. There is an intentional error on the */ 01380000
/* IF count so that an error message can be produced. */ 01390000
/*-----*/ 01400000
21 1 0 DECLARE CONTROLLED_SET( 36) FIXED BINARY(15) 01410000 1
INIT( 0, 3, 01420000
0, 1, 13, 24, 0, 2, 01430000
14, 01440000
13, 23, 0, 0, 0, 0, 1, 0, 14, 01450000
0, 7, 0, 4, 1, 01460000
2, 3, 01470000
2, 4, 0, 0, 1, 0, 01480000
2, 13, 0, 2, 0); 01490000
    
```

```

/*=====*/ 01510000
/* SAMPLE will perform the following tasks:          */ 01520000
/*= 1) OPEN the input dataset                          */ 01530000
/*= 2) READ each record and, for each record,         */ 01540000
/*= a) Extract a character string that meets the PL/I  */ 01550000
/*= definition of a word.                             */ 01560000
/*= b) If the word also appears in the list of interesting */ 01570000
/*= words, record its presence by incrementing a counter. */ 01580000
/*= 3) Report on the number of appearances of the words that */ 01590000
/*= actually appeared in the dataset.                 */ 01600000
/*= 4) DISPLAY a message if the count does not match the count */ 01610000
/*= of PL/I statement keywords in this program.       */ 01620000
/*=====*/ 01630000
01640000
/*-----*/ 01650000
/* Describe the action to take on selected exceptional conditions. */ 01660000
/*-----*/ 01670000
01680000
/*-----*/ 01690000
/* If the file has not been properly defined, tell them about it. */ 01700000
/*-----*/ 01710000
22 1 0 ON UNDEFINEDFILE(SOURCE) 01720000
      BEGIN; 01730000
23 2 0   DISPLAY ('The input data set has not been defined. '); 01740000
24 2 0   STOP; 01750000
25 2 0   END; 01760000
01770000
/*-----*/ 01780000
/* When the file has been processed indicate "no record read." */ 01790000
/*-----*/ 01800000
26 1 0 ON ENDFILE(SOURCE) 01810000
      BEGIN; 01820000
27 2 0   RECORD_READ = FALSE; 01830000
28 2 0   END; 01840000
01850000
/*-----*/ 01860000
/* If any other errors occur, write a message and terminate. */ 01870000
/*-----*/ 01880000
29 1 0 ON ERROR 01890000
      BEGIN; 01900000
30 2 0   ON ERROR SYSTEM; 01910000
31 2 0   DISPLAY ('Unspecified error occurred. ONCODE=' || ONCODE ); 01920000
32 2 0   STOP; 01930000
33 2 0   END; 01940000
01950000
/*-----*/ 01960000
/* Prepare the input dataset for processing -- mark it as open. */ 01970000
/*-----*/ 01980000
34 1 0 OPEN FILE(SOURCE) INPUT; 01990000

```

```

/*=====*/ 02000000
/* Count the use of PL/I statements in each record of the */ 02010000
/* input data set. */ 02020000
/*=====*/ 02030000
02040000
/*-----*/ 02050000
/* Read the first record of the input dataset. */ 02060000
/*-----*/ 02070000
35 1 0 RECORD_READ = TRUE; 02080000
36 1 0 READ FILE(SOURCE) INTO (RECORD); 02090000
02100000
/*-----*/ 02110000
/* Process the first and all succeeding records. */ 02120000
/*-----*/ 02130000
37 1 0 DO WHILE (RECORD_READ); 02140000
02150000
/* Set the "last character" position to the left margin */ 02160000
38 1 1 LAST_CHAR POSN = LEFT_MARGIN; 02170000
/* Use NEXT_WORD to extract the first word from this record. */ 02180000
39 1 1 WORD = NEXT_WORD(RECORD); 02190000
02200000
/*-----*/ 02210000
/* Extract words from this record until no more remain. */ 02220000
/*-----*/ 02230000
40 1 1 DO WHILE (WORD ^= ''); 02240000
/* Use LOOKUP_WORD to find its position in the table. */ 02250000
41 1 2 WORD_INDEX = LOOKUP_WORD(WORD); 02260000
02270000
/*-----*/ 02280000
/* If the word is in the list, count it. */ 02290000
/*-----*/ 02300000
42 1 2 IF WORD_INDEX ^= 0 THEN 02310000
WORD_COUNT(WORD_INDEX) = WORD_COUNT(WORD_INDEX) + 1; 02320000
43 1 2 ELSE; 02330000
/* Get the next word from the record. */ 02340000
44 1 2 WORD = NEXT_WORD(RECORD); 02350000
45 1 2 END; 02360000
02370000
/*-----*/ 02380000
/* Read the next record from the input data set. */ 02390000
/*-----*/ 02400000
46 1 1 READ FILE(SOURCE) INTO (RECORD); 02410000
47 1 1 END; 02420000
02430000
/*-----*/ 02440000
/* Input from the data set is exhausted. CLOSE it. */ 02450000
/*-----*/ 02460000
48 1 0 CLOSE FILE(SOURCE); 02470000

```

```

/*=====*/ 02490000
/* The report that details and summarizes the use of word in the */ 02500000
/* WORD_TABLE is prepared in this section. */ 02510000
/*=====*/ 02520000
02530000
49 1 0 PUT SKIP LIST (' ***** '); 02540000
50 1 0 PUT SKIP LIST (' *** Word-use Report *** '); 02550000
51 1 0 PUT SKIP LIST (' ***** '); 02560000
52 1 0 PUT SKIP LIST (' -count- --word-- '); 02570000
02580000
/*-----*/ 02590000
/* Review the activity for each word in the list. */ 02600000
/*-----*/ 02610000
53 1 0 DO WORD_INDEX = 1 TO 36; 02620000 1
02630000
/*-----*/ 02640000
/* If the word was used then display the word and its use-count. */ 02650000
/*-----*/ 02660000
54 1 1 IF WORD_COUNT(WORD_INDEX) > 0 THEN 02670000
PUT SKIP EDIT (WORD_COUNT(WORD_INDEX), 02680000
WORD_TABLE(WORD_INDEX)) 02690000
(F(6), X(6),A); 02700000
55 1 1 ELSE; 02710000
02720000
/*-----*/ 02730000
/* If there was a discrepancy between what was counted and what */ 02740000
/* was expected then display a warning message and remember that */ 02750000
/* it had occurred. */ 02760000
/*-----*/ 02770000
56 1 1 IF WORD_COUNT(WORD_INDEX) /= CONTROLLED_SET(WORD_INDEX) THEN 02780000
DO; 02790000
57 1 2 PUT SKIP EDIT ((12)'-', 02800000
'The previous value should have been', 02810000
CONTROLLED_SET(WORD_INDEX)) 02820000
(A, A, F(6)); 02830000
58 1 2 DISCREPANCY_OCCURRED = TRUE; 02840000
59 1 2 END; 02850000
60 1 1 ELSE; 02860000
61 1 1 END; 02870000
02880000

```

```

/*=====*/ 02890000
/* Summarize word activity on this input dataset. */ 02900000
/*=====*/ 02910000
02920000
62 1 0 PUT SKIP(2) LIST ('There were ' || SUM(WORD_COUNT) 02930000
|| ' references to ' || 36 02940000 1
|| ' words. '); 02950000
02960000
/*-----*/ 02970000
/* If a discrepancy between one of the counts and the expected */ 02980000
/* counts occurred then display a warning message. */ 02990000
/*-----*/ 03000000
63 1 0 IF DISCREPANCY_OCCURRED THEN 03010000
PUT SKIP(2) LIST ('There was a discrepancy in at least one of' 03020000
|| ' the word-counts. '); 03030000
64 1 0 ELSE; 03040000

```

```

/*==== NEXT_WORD =====*/ 03050000
/*=====*/ 03060000
/*== ==*/ 03070000
/*== Extract a word from the argument string that is passed. ==*/ 03080000
/*== Return it as CHAR(31) VARYING. ==*/ 03090000
/*== ==*/ 03100000
/*== Ignore PL/I comments and constants (strings surrounded by ==*/ 03110000
/*== single quotes (')). Comments and constants can not be ==*/ 03120000
/*== continued but must be complete in the argument string. ==*/ 03130000
/*== ==*/ 03140000
/*== If no more words exist then a null character string will ==*/ 03150000
/*== be returned. ==*/ 03160000
/*== ==*/ 03170000
/*=====*/ 03180000
/*=====*/ 03190000
03200000
65 1 0 NEXT_WORD: PROCEDURE(DATA_RECORD) RETURNS(CHAR(31) VARYING); 03210000
03220000
66 2 0 DECLARE DATA_RECORD CHAR(*) VARYING; 03230000
67 2 0 DECLARE DATA_WORD CHAR(31) VARYING; 03240000
68 2 0 DECLARE NEXT_CHARACTER CHAR(1); 03250000
69 2 0 DECLARE LENGTH_OF_STRING FIXED BINARY(15); 03260000
03270000
70 2 0 DECLARE NEXT_CHAR_POSN FIXED BINARY(15); 03280000
03290000
/*=====*/ 03300000
/*= LAST_CHAR POSN remembers, from call to call, the point where= */ 03310000
/*= the search for additional words will start. Management of = */ 03320000
/*= its value is a key concern to this function. = */ 03330000
/*= = */ 03340000
/*= Comments and constants in the argument string will be = */ 03350000
/*= ignored. If a character is found that is a legitimate PL/I = */ 03360000
/*= "first-character" then a word is assumed to follow. It = */ 03370000
/*= will be built by concatenating (suffixing) additional, = */ 03380000
/*= legitimate "next-characters." = */ 03390000
/*=====*/ 03400000
```

```

/*=====*/ 03410000
/*= Scan each character in the record. Start at the position = */ 03420000
/*= where scanning last terminated (LAST_CHAR_POSN) and = */ 03430000
/*= continue until the end of a word or the end of the record = */ 03440000
/*= is reached. = */ 03450000
/*=====*/ 03460000
03470000
71 2 0 DATA_WORD = ''; 03480000
72 2 0 DO NEXT_CHAR_POSN = LAST_CHAR_POSN TO RIGHT_MARGIN 03490000
WHILE (DATA_WORD = ''); 03500000
73 2 1 NEXT_CHARACTER = SUBSTR(DATA_RECORD, NEXT_CHAR_POSN, 1); 03510000
74 2 1 SELECT (NEXT_CHARACTER); 03520000
03530000
75 2 2 WHEN ('/') 03540000
03550000
/*-----*/ 03560000
/* If this turns out to be a comment then skip over it. */ 03570000
/*-----*/ 03580000
DO; 03590000
76 2 3 IF SUBSTR(DATA_RECORD, NEXT_CHAR_POSN, 2) = '/*' THEN 03600000
NEXT_CHAR_POSN = NEXT_CHAR_POSN + 3 03610000
+ INDEX(SUBSTR(DATA_RECORD, NEXT_CHAR_POSN+2), '/*'); 03620000
77 2 3 ELSE; 03630000
78 2 3 END; 03640000
03650000
79 2 2 WHEN ('''') 03660000
03670000
/*-----*/ 03680000
/* Skip over the constant. */ 03690000
/*-----*/ 03700000
NEXT_CHAR_POSN = NEXT_CHAR_POSN 03710000
+ INDEX(SUBSTR(DATA_RECORD, NEXT_CHAR_POSN+1), '''); 03720000
```

```

    /*=====*/ 03730000
    /*= This may be the start of a word. Extract it if so. =*/ 03740000
    /*=====*/ 03750000
80  2  2  OTHERWISE 03760000
    03770000
    03780000
    /*-----*/ 03790000
    /* This may be the start of a word. */ 03800000
    /*-----*/ 03810000
    DO; 03820000
    03830000
    /*-----*/ 03840000
    /* If the next character is not acceptable as the first */ 03850000
    /* character of a word then do nothing further -- our */ 03860000
    /* enclosing DO will step to the next character for */ 03870000
    /* further checking. */ 03880000
    /*-----*/ 03890000
81  2  3  IF INDEX(WORD_FIRST_CHARACTERS, NEXT_CHARACTER) = 0 THEN; 03900000
82  2  3  ELSE 03910000
    03920000
    /*-----*/ 03930000
    /* This is the start of a word. Collect the rest of it*/ 03940000
    /*-----*/ 03950000
    03960000
    DO; 03970000
    DATA_WORD = NEXT_CHARACTER; 03980000
    03990000
    /*-----*/ 04000000
    /* Build up DATA_WORD by iteratively appending */ 04010000
    /* characters from the input argument string. Do it */ 04020000
    /* as long as the characters are acceptable PL/I */ 04030000
    /* "next=characters." */ 04040000
    /*-----*/ 04050000
    04060000
84  2  4  DO NEXT_CHAR_POSN = NEXT_CHAR_POSN+1 TO RIGHT_MARGIN 04070000
    WHILE (INDEX(WORD_NEXT_CHARACTERS, 04080000
    SUBSTR(DATA_RECORD, NEXT_CHAR_POSN,1)) /= 0); 04090000
85  2  5  DATA_WORD = DATA_WORD 04100000
    || SUBSTR(DATA_RECORD, NEXT_CHAR_POSN,1); 04110000
86  2  5  END; 04120000
87  2  4  LAST_CHAR_POSN = NEXT_CHAR_POSN + 1; 04130000
88  2  4  END; 04140000
89  2  3  END; 04150000
90  2  2  END; /* End of the SELECT (NEXT_CHARACTER) statement */ 04160000
91  2  1  END; /* End of the DO that tries to find a word */ 04170000
    04180000
92  2  0  RETURN (DATA_WORD); 04190000
93  2  0  END; 04200000
    
```

```

    /*==== LOOKUP_WORD =====*/ 04210000
    /*====*/ 04220000
    /*== ==*/ 04230000
    /*== Find the word in the WORD_TABLE that matches the ==*/ 04240000
    /*== argument string (CHAR(*) VARYING) and return the ==*/ 04250000
    /*== position of that word (its subscript) to the ==*/ 04260000
    /*== invoker (FIXED BINARY(15)). ==*/ 04270000
    /*== ==*/ 04280000
    /*== If the word does not exist in the list a 0 will be ==*/ 04290000
    /*== returned. ==*/ 04300000
    /*== ==*/ 04310000
    /*====*/ 04320000
    /*====*/ 04330000
    /*====*/ 04340000
    94 1 0 LOOKUP_WORD: PROCEDURE(DATA_WORD) RETURNS(FIXED BINARY(15)); 04350000
    /*====*/ 04360000
    95 2 0 DECLARE DATA_WORD CHAR(*) VARYING; 04370000
    96 2 0 DECLARE WORD_NUMBER FIXED BINARY(15); 04380000
    /*====*/ 04390000
    /*== A sequential search is used to locate the required word. ==*/ 04400000
    /*== WORD_INDEX_TABLE is used to start the search at the first ==*/ 04410000
    /*== word in the list that has the same first character. ==*/ 04420000
    /*====*/ 04430000
    97 2 0 WORD_NUMBER = WORD_INDEX_TABLE /* Subscript is on next line */ 04440000
    (INDEX(WORD_FIRST_CHARACTERS, SUBSTR(DATA_WORD,1,1))); 04450000
    /*====*/ 04460000
    /*-----*/ 04470000
    /* Search words in the WORD_TABLE until the word is found or is */ 04480000
    /* determined to be not a part of the list -- its index number */ 04490000
    /* (WORD_NUMBER) is zero. */ 04500000
    /*-----*/ 04510000
    98 2 0 DO WORD_NUMBER = WORD_NUMBER BY 1 04520000
    UNTIL ( WORD_TABLE(WORD_NUMBER) = DATA_WORD 04530000
    | WORD_NUMBER = 0); 04540000
    /*-----*/ 04550000
    /*-----*/ 04560000
    /* If the word in the WORD_TABLE is alphabetically greater */ 04570000
    /* than the argument word then a match cannot be found. Set */ 04580000
    /* the WORD_NUMBER to 0 to indicate a non=match situation. */ 04590000
    /*-----*/ 04600000
    99 2 1 IF WORD_TABLE(WORD_NUMBER) > DATA_WORD THEN 04610000
    WORD_NUMBER = 0; 04620000
    100 2 1 ELSE; 04630000
    101 2 1 END; 04640000
    102 2 0 RETURN (WORD_NUMBER); 04650000
    103 2 0 END; 04660000
    104 1 0 END; 04670000
    04680000
    06230000
    
```





## STORAGE REQUIREMENTS

<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>			
BLOCK, SECTION OR STATEMENT	TYPE	LENGTH	(HEX)	DSA	SIZE	(HEX)
*SAMPLE1	PROGRAM CSECT	4444	115C			
*SAMPLE2	STATIC CSECT	2880	B40			
SAMPLE	PROCEDURE BLOCK	2298	8FA	1208	4B8	
BLOCK 2 STMT 22	ON UNIT	170	AA	208	D0	
BLOCK 3 STMT 26	ON UNIT	162	A2	216	D8	
BLOCK 4 STMT 29	ON UNIT	304	130	288	120	
NEXT_WORD	PROCEDURE BLOCK	1056	420	368	170	
LOOKUP_WORD	PROCEDURE BLOCK	448	1C0	256	100	

Storage requirements. This table gives the main storage requirements for the program. These quantities do not include the main storage required by the library subroutines that will be included by the linkage editor or loaded dynamically during execution.

- 5** Name of the block, section, or number of the statement in the program.
- 6** Description of the block, section, or statement.
- 7** Length in bytes of the storage areas in both decimal and hexadecimal notation.
- 8** Length in bytes of the dynamic storage area (DSA) in both decimal and hexadecimal notation.

## EXTERNAL SYMBOL DICTIONARY

<b>1</b> SYMBOL	<b>2</b> TYPE	<b>3</b> ID	<b>4</b> ADDR	<b>5</b> LENGTH
CEESTART	SD	0001	000000	000080
*SAMPLE1	SD	0002	000000	00115C
*SAMPLE2	SD	0003	000000	000B40
CEEMAIN	WX	0004	000000	
CEEMAIN	SD	0005	000000	000010
IBMRINP1	ER	0006	000000	
CEEFMAIN	WX	0007	000000	
CEEBETBL	ER	0008	000000	
CEERootA	ER	0009	000000	
CEEOP1PI	ER	000A	000000	
CEESG010	ER	000B	000000	
IBMSEATA	ER	000C	000000	
IELCGOG	SD	000D	000000	0000AE
IELCGOH	SD	000E	000000	0000A0
IELCGOC	SD	000F	000000	00007C
IELCGMY	SD	0010	000000	0000A4
IELCGCY	SD	0011	000000	00007E
IBMSSIOA	ER	0012	000000	
IBMSASCA	ER	0013	000000	
IBMSCEDB	ER	0014	000000	
IBMSCHFD	ER	0015	000000	
IBMSCHXH	WX	0016	000000	
IBMSCWDH	ER	0017	000000	
IBMSEOCA	ER	0018	000000	
IBMSJDSA	ER	0019	000000	
IBMSOCLA	ER	001A	000000	
IBMSOCLC	WX	001B	000000	
IBMSRIOA	ER	001C	000000	
IBMSSEOA	ER	001D	000000	
IBMSSIOE	WX	001E	000000	
IBMSSIOT	WX	001F	000000	
IBMSSL0A	ER	0020	000000	
IBMSSPLA	ER	0021	000000	
IBMSSXCA	ER	0022	000000	
IBMSSXCB	WX	0023	000000	
IBMSSIST	WX	0024	000000	
CEEUOPT	SD	0025	000000	0004D0
PLIXOPT	SD	0026	000000	000066
PLIXOPT*	SD	0027	000000	00002C
PLIXOPT+	LD		000004	
PLIXOPT-	LD		000020	
PLIXOPT-	ER	0028	000000	
SAMPLE	LD		000008	
SAMPLE	ER	0029	000000	
SAMPLE*	SD	002A	000000	000020
SAMPLE+	LD		000000	
SOURCE	SD	002B	000000	00001C

## External symbol dictionary

**1** List of all the external symbols that make up the object module.

**2** Type of external symbol, as follows:

CM Common area  
ER External reference  
LD Label definition  
PR Pseudo-register  
SD Section definition  
WX Weak external reference

Full definitions of all these terms are given in "External symbol dictionary" in the main text.

**3** All entries, except LD type entries, are identified by a hexadecimal number.

**4** Address (in hexadecimal) of LD type entries.

**5** Length in bytes (in hexadecimal) of SD, CM, and PR type entries.

SOURCE	PR	002C	000000	000004
SOURCE*	SD	002D	000000	000020
SOURCE+	LD		000000	
SYSPINT	SD	002E	000000	000020
SYSPINT*	SD	002F	000000	000020
SYSPINT+	LD		000000	

<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>
		STATIC INTERNAL STORAGE MAP	0000C8	60000006	FED
			0000CC	58000009	FED
			0000D0	5800000C	FED
			0000D4	58000023	FED
			0000D8	2800	DED..WORD
			0000DA	2401	DED..FALSE
			0000DC	0002	CONSTANT
			0000DE	0048	CONSTANT
			0000E0	0000	CONSTANT
			0000E2	0001	CONSTANT
			0000E4	0003	CONSTANT
			0000E6	0005	CONSTANT
			0000E8	000A	CONSTANT
			0000EA	000D	CONSTANT
			0000EC	000E	CONSTANT
			0000EE	0012	CONSTANT
			0000F0	0013	CONSTANT
			0000F2	0016	CONSTANT
			0000F4	0018	CONSTANT
			0000F6	001A	CONSTANT
			0000F8	001E	CONSTANT
			0000FA	0021	CONSTANT
			0000FC	0022	CONSTANT
			0000FE	0017	CONSTANT
			000100	0007	CONSTANT
			000102	0004	CONSTANT
			000104	0024	CONSTANT
			000106	0009	CONSTANT
			000108	4040402020202020 202020202120	CONSTANT
			000116	001C	CONSTANT
			000118	001D	CONSTANT
			00011A		
			000120	00000000000001B4	LOCATOR..CONTROLLED_SET
			000128	00000000000001C4	LOCATOR..WORD_INDEX_TABLE
			000130	00000000000001D4	LOCATOR..WORD_TABLE
			000138	00000000001F8000	LOCATOR..WORD
			000140	00000535001E0000	LOCATOR..WORD_NEXT_CHARACTERS
			000148	00000518001D0000	LOCATOR..WORD_FIRST_CHARACTERS
			000150	0000000000010000	LOCATOR..FALSE
			000158	0000000000798000	LOCATOR..RECORD
			000160	0000000000080000	LOCATOR..DATA_RECORD
			000168	0080000091102000	CONSTANT
			000170	000000000200007B	RECORD_DESCRIPTOR
			000178	000003D800190000	LOCATOR
			000180	000003F100190000	LOCATOR
			000188	0000040A00140000	LOCATOR
			000190	0000000000340000	LOCATOR
			000198	00000000003B0000	LOCATOR
			0001A0	000004AE00280000	LOCATOR
			0001A8	00000000002D0000	LOCATOR
			0001B0	91E091E0	CONSTANT

Static internal storage map.  
 This is a storage map of the static control section for the program. This control section is the third standard entry in the external symbol dictionary.

**1** Six-digit offset (in hexadecimal)

**2** Text (in hexadecimal)

**3** Comment indicating type of item to which the text refers. A comment appears only against the first line of the text for an item.

```

5688-235 IBM PL/I for MVS & VM          /* PL/I Sample Program: Used to verify product installation */PAGE 37
0001B4 0000000200000002  DESCRIPTOR          40
0000002400000001          0002A6 C3C1D3D340404040  CONSTANT
0001C4 0000000200000002  DESCRIPTOR          40
0000001A00000001          0002AF C3D3D6E2C5404040  CONSTANT
0001D4 0000000900000009  DESCRIPTOR          40
0000002500000001          0002B8 C4C3D34040404040  CONSTANT
00090000          40
0001E8 00000001          CONSTANT          0002C1 C4C5C3D3C1D9C540  CONSTANT
0001EC 0010000000601800  CONSTANT          40
00000000          0002CA C4C5C6C1E4D3E340  CONSTANT
0001F8 00000002          CONSTANT          40
0001FC 00000003          CONSTANT          0002D3 C4C9E2D7D3C1E840  CONSTANT
000200 0000001F          CONSTANT          40
000204 00000000          A..PLIXOPT          0002DC C4D6404040404040  CONSTANT
000208 00000000          A..DCLCB          40
00020C 00000000          A..DCLCB          0002E5 C5D3E2C540404040  CONSTANT
000210 000001E8          A..CONSTANT          40
000214 00000000          A..DCLCB          0002EE C5D5C44040404040  CONSTANT
00000000          40
000218 000001EC          A..CONSTANT          40
00021C 00000000          OMITTED ARGUMENT  0002F7 C5D5E3D9E8404040  CONSTANT
000220 00000000          OMITTED ARGUMENT  40
000224 80000000          OMITTED ARGUMENT  000300 C6D9C5C540404040  CONSTANT
000228 00000000          A..DCLCB          40
00022C 00000168          A..CONSTANT          000309 C7C5D5C5D9C9C340  CONSTANT
000230 00000000          A..RD          40
000234 00000000          OMITTED ARGUMENT  000312 C7C5E34040404040  CONSTANT
000238 00000000          OMITTED ARGUMENT  40
00023C 80000000          OMITTED ARGUMENT  00031B C7D6404040404040  CONSTANT
000240 00000000          A..LOCATOR          40
000244 80000000          A..TEMP          000324 C7D6E3D640404040  CONSTANT
000248 00000000          A..LOCATOR          40
00024C 80000000          A..WORD_INDEX     00032D C9C6404040404040  CONSTANT
000250 000001E8          A..CONSTANT          40
000254 00000000          A..DCLCB          000336 D3C5C1E5C5404040  CONSTANT
000258 80000000          OMITTED ARGUMENT  40
00025C 00000000          A..DCLCB          00033F D3C9E2E340404040  CONSTANT
000260 00000000          A..TEMP          40
000264 800001E8          A..CONSTANT          000348 D3D6C3C1E3C54040  CONSTANT
000268 00000000          A..DCLCB          40
00026C 00000000          A..TEMP          000351 D6D5404040404040  CONSTANT
000270 800001F8          A..CONSTANT          40
000274 00000000          A..LOCATOR          00035A D6D7C5D540404040  CONSTANT
000278 000000E2          A..CONSTANT          40
00027C 000000BE          A..DED..WORD_COUNT 000363 D7D9D6C340404040  CONSTANT
000280 00000000          A..TEMP          40
000284 800000E0          A..CONSTANT          00036C D7D9D6C3C5C4E4D9  CONSTANT
000288 800001A0          A..CONSTANT          C5
00028C 0D800000          CONSTANT          000375 D9C5C1C440404040  CONSTANT
000290 80000000          A..TEMP          40
000294 C1D3D3D6C3C1E3C5  CONSTANT          00037E D9C5E3E4D9D54040  CONSTANT
40          40
00029D C2C5C7C9D5404040  CONSTANT          000387 D9C5E5C5D9E34040  CONSTANT

```

```

5688-235 IBM PL/I for MVS & VM          /* PL/I Sample Program: Used to verify product installation          */PAGE 38
      40                                96A3408285859540
000390 D9C5E6D9C9E3C540          CONSTANT          848586899585844B
      40                                0004D6 4040202020202021          CONSTANT
000399 E2C5D3C5C3E34040          CONSTANT          20
      40                                0004DF E495A29785838986          CONSTANT
0003A2 E2C9C7D5C1D34040          CONSTANT          8985844085999996
      40                                9940968383A49999
0003AB E2E3D6D740404040          CONSTANT          85844B4040D6D5C3
      40                                D6C4C57E
0003B4 E3C8C5D540404040          CONSTANT          000503 615C          CONSTANT
      40                                000505 5C61          CONSTANT
0003BD E6C1C9E340404040          CONSTANT          000507 7D          CONSTANT
      40                                000508 0C16000000000A4C          STATIC ONCB
0003C6 E6C8C5D540404040          CONSTANT          000510 0C96000000000000          STATIC ONCB
      40                                000518 C1C2C3C4C5C6C7C8          INITIAL VALUE..WORD_FIRST_CHAR
0003CF E6D9C9E3C5404040          CONSTANT          C9D1D2D3D4D5D6D7
      40                                D8D9E2E3E4E5E6E7
0003D8 405C5C5C5C5C5C5C          CONSTANT          E8E97C7B5B
      5C5C5C5C5C5C5C5C          000535 C1C2C3C4C5C6C7C8          INITIAL VALUE..WORD_NEXT_CHAR
      5C5C5C5C5C5C5C5C          C9D1D2D3D4D5D6D7
      40                                D8D9E2E3E4E5E6E7
0003F1 405C5C5C40E69699          CONSTANT          E8E96D7C7B5B
      8460A4A28540D985          000558 00000000          SYMBOL TABLE ELEMENT
      979699A3405C5C5C          00055C 00000000          CONSTANT
      40                                000560 00000000          SYMBOL TABLE ELEMENT
00040A 40608396A495A360          CONSTANT          000564 00000604          SYMBOL TABLE ELEMENT
      4040406060A69699          000568 00000624          SYMBOL TABLE ELEMENT
      84606040          00056C 00000648          SYMBOL TABLE ELEMENT
00041E 6060606060606060          CONSTANT          000570 00000664          SYMBOL TABLE ELEMENT
      60606060          000574 80000000          SYMBOL TABLE ELEMENT
00042A E3888540979985A5          CONSTANT          000578 00000680          SYMBOL TABLE ELEMENT
      8996A4A240A58193          00057C 0000069C          SYMBOL TABLE ELEMENT
      A48540A28896A493          000580 000006B4          SYMBOL TABLE ELEMENT
      84408881A5854082          000584 000006DC          SYMBOL TABLE ELEMENT
      858595          000588 00000704          SYMBOL TABLE ELEMENT
00044D E38885998540A685          CONSTANT          00058C 0000071C          SYMBOL TABLE ELEMENT
      998540          000590 00000734          SYMBOL TABLE ELEMENT
000458 4099858685998595          CONSTANT          000594 00000754          SYMBOL TABLE ELEMENT
      8385A240A39640          000598 00000774          SYMBOL TABLE ELEMENT
000467 404040F3F6          CONSTANT          00059C 0000079C          SYMBOL TABLE ELEMENT
00046C 40A6969984A24B          CONSTANT          0005A0 000007BC          SYMBOL TABLE ELEMENT
000473 E38885998540A681          CONSTANT          0005A4 000007DC          SYMBOL TABLE ELEMENT
      A24081408489A283          0005A8 000008D0          SYMBOL TABLE ELEMENT
      998597819583A840          0005AC 000008EC          SYMBOL TABLE ELEMENT
      89954081A3409385          0005B0 00000000          SYMBOL TABLE ELEMENT
      81A2A34096958540          0005B4 00000000          SYMBOL TABLE ELEMENT
      968640A3888540A6          0005B8 00000000          CONSTANT
      969984608396A495          0005BC 00000558          SYMBOL TABLE ELEMENT
      A3A24B          0005C0 00000000          CONSTANT
0004AE E3888540899597A4          CONSTANT          0005C4 00000564          SYMBOL TABLE ELEMENT
      A3408481A38140A2          0005C8 00000000          CONSTANT
      85A3408881A24095          0005CC 00000564          SYMBOL TABLE ELEMENT

```

```

5688-235 IBM PL/I for MVS & VM          /* PL/I Sample Program: Used to verify product installation          */PAGE 39
0005D0 00000000 CONSTANT 0004E3D9E4C50000
0005D4 00000564 SYMBOL TABLE ELEMENT 000734 85000001000000BE SYMBOL TABLE..RIGHT_MARGIN
0005D8 000007F4 SYMBOL TABLE ELEMENT 0000011A00000000
0005DC 00000814 SYMBOL TABLE ELEMENT 000CD9C9C7C8E36D
0005E0 00000838 SYMBOL TABLE ELEMENT D4C1D9C7C9D50000
0005E4 00000858 SYMBOL TABLE ELEMENT 000754 85000001000000BE SYMBOL TABLE..LEFT_MARGIN
0005E8 00000874 SYMBOL TABLE ELEMENT 0000011C00000000
0005EC 00000000 CONSTANT 000BD3C5C6E36DD4
0005F0 00000564 SYMBOL TABLE ELEMENT C1D9C7C9D5000000
0005F4 00000894 SYMBOL TABLE ELEMENT 000774 81000001000000DA SYMBOL TABLE..DISCREPANCY_OCCU
0005F8 000008B4 SYMBOL TABLE ELEMENT 000000F800000000
0005FC 00000000 CONSTANT 0014C4C9E2C3D9C5
000600 00000564 SYMBOL TABLE ELEMENT D7C1D5C3E86DD6C3
000604 81000101000000BE SYMBOL TABLE..CONTROLLED_SET C3E4D9D9C5C40000
000000C000000000 00079C 85000001000000BE SYMBOL TABLE..LAST_CHAR_POSN
000EC3D6D5E3D9D6 0000011E00000000
D3D3C5C46DE2C5E3 000ED3C1E2E36DC3
000624 81000101000000BE SYMBOL TABLE..WORD_INDEX_TABLE C8C1D96DD7D6E2D5
000000C800000000 0007BC 81000001000000DA SYMBOL TABLE..RECORD_READ
0010E6D6D9C46DC9 0000010000000000
D5C4C5E76DE3C1C2 000BD9C5C3D6D9C4
D3C50000 6DD9C5C1C4000000
000648 81000101000000BE SYMBOL TABLE..WORD_COUNT 0007DC 81000001000000D8 SYMBOL TABLE..RECORD
000000D000000000 0000010800000000
000AE6D6D9C46DC3 0006D9C5C3D6D9C4
D6E4D5E3 0007F4 85000002000000BE SYMBOL TABLE..NEXT_CHAR_POSN
000664 81000101000000BC SYMBOL TABLE..WORD_TABLE 000000D000000000
000000D800000000 000ED5C5E7E36DC3
000AE6D6D9C46DE3 C8C1D96DD7D6E2D5
C1C2D3C5 000814 85000002000000BE SYMBOL TABLE..LENGTH_OF_STRING
000680 85000001000000BE SYMBOL TABLE..WORD_INDEX 000000D200000000
0000011800000000 0010D3C5D5C7E3C8
000AE6D6D9C46DC9 6DD6C66DE2E3D9C9
D5C4C5E7 D5C70000
00069C 81000001000000D8 SYMBOL TABLE..WORD 000838 81000002000000BC SYMBOL TABLE..NEXT_CHARACTER
000000E000000000 000000B800000000
0004E6D6D9C40000 000ED5C5E7E36DC3
0006B4 01000000000000BC SYMBOL TABLE..WORD_NEXT_CHARAC C8C1D9C1C3E3C5D9
0000014000000000 000858 81000002000000D8 SYMBOL TABLE..DATA_WORD
0014E6D6D9C46DD5 000000C000000000
C5E7E36DC3C8C1D9 0009C4C1E3C16DE6
C1C3E3C5D9E20000 D6D9C400
0006DC 01000000000000BC SYMBOL TABLE..WORD_FIRST_CHARA000874 A1000002000000D8 SYMBOL TABLE..DATA_RECORD
0000014800000000 0000010000000000
0015E6D6D9C46DC6 000BC4C1E3C16DD9
C9D9E2E36DC3C8C1 C5C3D6D9C4000000
D9C1C3E3C5D9E200 000894 85000002000000BE SYMBOL TABLE..WORD_NUMBER
000704 81000001000000DA SYMBOL TABLE..FALSE 000000C000000000
000000E800000000 000BE6D6D9C46DD5
0005C6C1D3E2C500 E4D4C2C5D9000000
00071C 81000001000000DA SYMBOL TABLE..TRUE 0008B4 A1000002000000D8 SYMBOL TABLE..DATA_WORD
000000F000000000 000000D000000000

```



## TABLES OF OFFSETS AND STATEMENT NUMBERS

	WITHIN PROCEDURE SAMPLE														
OFFSET (HEX)	0	348	358	368	370	37E	388	39E	3AE	3BA	410	432	462	48A	492
STATEMENT NO.	1	22	26	29	34	35	36	37	38	39	40	41	42	43	44
OFFSET (HEX)	4E8	4F0	506	50E	51C	55C	59C	5DC	61C	62E	63A	642	6DA	6E2	6FA
STATEMENT NO.	45	46	47	48	49	50	51	52	53	54	53	54	55	56	57
OFFSET (HEX)	78A	794	79C	7A4	7A8	7B4	7CC	7CC	86E	8D2	8DA				
STATEMENT NO.	58	59	60	61	54	61	53	62	63	64	104				
	WITHIN ON UNIT BLOCK 2														
OFFSET (HEX)	0	76	84	92											
STATEMENT NO.	22	23	24	25											
	WITHIN ON UNIT BLOCK 3														
OFFSET (HEX)	0	80	8A												
STATEMENT NO.	26	27	28												
	WITHIN ON UNIT BLOCK 4														
OFFSET (HEX)	0	90	98	10A	118										
STATEMENT NO.	29	30	31	32	33										
	WITHIN PROCEDURE NEXT_WORD														
OFFSET (HEX)	0	BC	C8	100	11A	124	134	1C0	1C8	1D0	24C	254	292	29A	2AC
STATEMENT NO.	65	71	72	73	74	75	76	77	78	79	80	81	82	83	84
OFFSET (HEX)	328	386	38A	38E	3A4	3B8	3BC	3C0	3C0	3C4	3DC	414			
STATEMENT NO.	85	86	84	86	87	88	89	90	90	91	92	93			
	WITHIN PROCEDURE LOOKUP_WORD														
OFFSET (HEX)	0	98	DA	E2	132	13A	13E	146	148	15E	176	182	184	18C	190
STATEMENT NO.	94	97	98	99	100	101	98	101	98	101	98	101	98	101	102
OFFSET (HEX)	1B4														
STATEMENT NO.	103														





MVS/DFP VERSION 3 RELEASE 3 LINKAGE EDITOR 10:03:27 FRI JAN 29, 1993  
 JOB IEL111VP STEP IVP PROCEDURE LKED  
 INVOCATION PARAMETERS - XREF,LIST **1**  
 ACTUAL SIZE=(317440,79872)  
 OUTPUT DATA SET SYS93029.T100323.RA000.IEL111VP.GOSET IS ON VOLUME PUB002

<b>3</b>			<b>2</b> CROSS REFERENCE TABLE							
CONTROL	SECTION		ENTRY							
NAME	ORIGIN	LENGTH	NAME	LOCATION	NAME	LOCATION	NAME	LOCATION	NAME	LOCATION
CEESTART	00	80								
CEEMAIN	80	10								
SYSPINT	90	20								
SOURCE	B0	1C								
CEEUOPT	D0	4D0								
PLIXOPT	5A0	66								
PLIXOPT*	608	2C								
			PLIXOPT+	60C	PLIXOPT-	628				
SAMPLE*	638	20								
			SAMPLE+	638						
SOURCE*	658	20								
			SOURCE+	658						
SYSPINT*	678	20								
			SYSPINT+	678						

Linkage editor listing.

- 1** Statement identifying the version and level of the linkage editor and giving the options as specified in the PARM parameter of the EXEC statement that invokes the linkage editor
- 2** Cross reference table, consisting of a module map and the cross reference table
- 3** The module map shows each control section and its associated entry points, if any, listed across the page. An asterisk in column 9 after a name beginning with "IBM" indicates a library subroutine obtained by automatic library call.
- 4** The cross reference table gives all the locations in a control section at which a symbol is reference. UNRESOLVED(W) identifies a weak external reference that has not been resolved.

*SAMPLE2	698	B40		
IELCGOG	11D8	AE		
IELCGOH	1288	A0		
IELCGOC	1328	7C		
IELCGMY	13A8	A4		
IELCGCY	1450	7E		
*SAMPLE1	14D0	115C		
			SAMPLE	14D8
CEEBETBL*	2630	1C		
CEEOP1PI*	2650	208		
CEER00TA*	2858	268		
CEESG010*	2AC0	64		
IBMRINP1*	2B28	24		
IBMSASCA*	2B50	14		
			IBMBASCA	2B50
IBMSCEDB*	2B68	14		
			IBMBCEDB	2B68
IBMSCHFD*	2B80	14		
			IBMBCHFD	2B80
IBMSEATA*	2B98	14		

IBMSSIOA*	2BB0	14	IBMBEATA	2B98						
IBMSCHXH*	2BC8	14	IBMBSIOA	2BB0						
IBMSCWDH*	2BE0	14	IBMBCHXH	2BC8						
			IBMBCWDH	2BE0						
NAME	ORIGIN	LENGTH	NAME	LOCATION	NAME	LOCATION	NAME	LOCATION	NAME	LOCATION
IBMSEOCA*	2BF8	14	IBMSEOCA	2BF8						
IBMSJDSA*	2C10	14	IBMBJDSA	2C10						
IBMSOCLA*	2C28	14	IBMBOCLA	2C28						
IBMSRIOA*	2C40	14	IBMBRIOA	2C40						
IBMSSEOA*	2C58	14	IBMBSEOA	2C58						
IBMSSIOE*	2C70	14	IBMBSIOE	2C70						
IBMSSIOT*	2C88	14	IBMBSIOT	2C88						
IBMSSLOA*	2CA0	14	IBMBSLOA	2CA0						
CEEARLU *	2CB8	140								
CEEBINT *	2DF8	8								
CEEBLLST*	2E00	5C	CEELLIST	2E10						
CEEBTRM *	2E60	180								
CEEP#CAL*	2FE0	120								
CEEP#INT*	3100	340								
CEEP#TRM*	3440	210								
IBMSOCLC*	3650	14	IBMBOCLC	3650						
IBMSSPLA*	3668	14	IBMBSPLA	3668						
IBMSSXCA*	3680	14	IBMBSXCA	3680						
CEEBPIRA*	3698	2E0	CEEINT	3698	CEEBPIRB	3698	CEEBPIRC	3698		
IBMSCEDF*	3978	14	IBMBCEDF	3978						
IBMSCEDX*	3990	14	IBMBCEDX	3990						
IBMSCEFX*	39A8	14	IBMBCEFX	39A8						
IBMSCEZB*	39C0	14	IBMBCEZB	39C0						
IBMSCEZF*	39D8	14	IBMBCEZF	39D8						
IBMSCEZX*	39F0	14	IBMBCEZX	39F0						
IBMSCHFE*	3A08	14	IBMBCHFE	3A08						
CEEPMATH*	3A20	18	IBMSMATH	3A20						
IBMSSXCB*	3A38	14	IBMBSXCB	3A38						
IBMSCHF*	3A50	14	IBMBCHF*	3A50						

NAME	ORIGIN	LENGTH	NAME	LOCATION	NAME	LOCATION	NAME	LOCATION	NAME	LOCATION
IBMSCHFP*	3A68	14	IBMBCHFP	3A68						
IBMSCHFY*	3A80	14	IBMBCHFY	3A80						
IBMSCHXD*	3A98	14	IBMBCHXD	3A98						
IBMSCHXE*	3AB0	14	IBMBCHXE	3AB0						
IBMSCHXF*	3AC8	14	IBMBCHXF	3AC8						
IBMSCHXP*	3AE0	14	IBMBCHXP	3AE0						
IBMSCHXY*	3AF8	14	IBMBCHXY	3AF8						
IBMSSIOB*	3B10	14	IBMBSIOB	3B10						
IBMSSIOC*	3B28	14	IBMBSIOC	3B28						
IBMSCWZH*	3B40	14	IBMBCWZH	3B40						
IBMSJDSB*	3B58	14	IBMBJDSB	3B58						
IBMSOCLB*	3B70	14	IBMBOCLB	3B70						
IBMSOCLD*	3B88	14	IBMBOCLD	3B88						
IBMSRIOB*	3BA0	14	IBMBRIOB	3BA0						
IBMSRIOC*	3BB8	14	IBMBRIOC	3BB8						
IBMSRIOD*	3BD0	14	IBMBRIOD	3BD0						
IBMSSIOD*	3BE8	14	IBMBSIOD	3BE8						
IBMSSLOB*	3C00	14	IBMBSLOB	3C00						
IBMSSPLB*	3C18	14	IBMBSPLB	3C18						
IBMSSPLC*	3C30	14	IBMBSPLC	3C30						
IBMSSXCC*	3C48	14	IBMBSXCC	3C48						
IBMSSXCD*	3C60	14	IBMBSXCD	3C60						

LOCATION	REFERS TO	SYMBOL	IN CONTROL SECTION
2C		CEEMAIN	CEEMAIN
74		CEEBETBL	CEEBETBL
84		*SAMPLE1	*SAMPLE1
62C		PLIXOPT	PLIXOPT

LOCATION	REFERS TO	SYMBOL	IN CONTROL SECTION
68		CEEFMAIN	4 \$UNRESOLVED(w)
78		CEERootA	CEERootA
88		IBMRINP1	IBMRINP1
640		SAMPLE	*SAMPLE1

LOCATION	REFERS TO	SYMBOL	IN CONTROL SECTION
660		SOURCE	SOURCE
69C		*SAMPLE1	*SAMPLE1
6A4		*SAMPLE1	*SAMPLE1
6AC		*SAMPLE1	*SAMPLE1
6B4		*SAMPLE1	*SAMPLE1
6BC		*SAMPLE1	*SAMPLE1
6C4		*SAMPLE1	*SAMPLE1
6CC		*SAMPLE1	*SAMPLE1
6D4		*SAMPLE1	*SAMPLE1
6DC		*SAMPLE1	*SAMPLE1
6E4		*SAMPLE1	*SAMPLE1

LOCATION	REFERS TO	SYMBOL	IN CONTROL SECTION
680		SYSPINT	SYSPINT
6A0		*SAMPLE1	*SAMPLE1
6A8		*SAMPLE1	*SAMPLE1
6B0		*SAMPLE1	*SAMPLE1
6B8		*SAMPLE1	*SAMPLE1
6C0		*SAMPLE1	*SAMPLE1
6C8		*SAMPLE1	*SAMPLE1
6D0		*SAMPLE1	*SAMPLE1
6D8		*SAMPLE1	*SAMPLE1
6E0		*SAMPLE1	*SAMPLE1
6E8		*SAMPLE1	*SAMPLE1

6EC	*SAMPLE1	*SAMPLE1	6F0	*SAMPLE1	*SAMPLE1
6F4	IELCGOG	IELCGOG	6F8	IELCGOH	IELCGOH
6FC	IELCGOC	IELCGOC	700	IELCGMY	IELCGMY
704	IELCGCY	IELCGCY	708	IBMSASCA	IBMSASCA
70C	IBMSCEDB	IBMSCEDB	710	IBMSCHFD	IBMSCHFD
714	IBMSCHXH	IBMSCHXH	718	IBMSCWDH	IBMSCWDH
71C	IBMSEOCA	IBMSEOCA	720	IBMSJDSA	IBMSJDSA
724	IBMSOCLA	IBMSOCLA	728	IBMSOCLC	IBMSOCLC
72C	IBMSRIOA	IBMSRIOA	730	IBMSSEOA	IBMSSEOA
734	IBMSSIOE	IBMSSIOE	738	IBMSSIOT	IBMSSIOT
73C	IBMSSLOA	IBMSSLOA	740	IBMSSPLA	IBMSSPLA
744	IBMSSXCA	IBMSSXCA	748	IBMSSXCB	IBMSSXCB
89C	PLIXOPT	PLIXOPT	8A0	SYSPINT	SYSPINT
8A4	SOURCE	SOURCE	8AC	SOURCE	SOURCE
8C0	SOURCE	SOURCE	8EC	SOURCE	SOURCE
8F4	SYSPINT	SYSPINT	900	SYSPINT	SYSPINT
BA4	*SAMPLE1	*SAMPLE1	BF0	SAMPLE*	SAMPLE*
C0C	PLIXOPT*	PLIXOPT*	C48	SOURCE*	SOURCE*
C4C	SYSPINT*	SYSPINT*	F70	*SAMPLE1	*SAMPLE1
F8C	*SAMPLE1	*SAMPLE1	FA8	*SAMPLE1	*SAMPLE1
106C	*SAMPLE1	*SAMPLE1	108C	*SAMPLE1	*SAMPLE1
10A8	*SAMPLE1	*SAMPLE1	10CC	*SAMPLE1	*SAMPLE1
1164	*SAMPLE1	*SAMPLE1	11D4	CEESTART	CEESTART
1320	IBMSSIST	\$UNRESOLVED (w)	1324	IBMSSEOA	IBMSSEOA
139C	IBMSSXCB	IBMSSXCB	13A0	IBMSSIST	\$UNRESOLVED (w)
14E0	*SAMPLE2	*SAMPLE2	14E8	*SAMPLE2	*SAMPLE2
14EC	*SAMPLE2	*SAMPLE2	1510	*SAMPLE2	*SAMPLE2
1DD4	*SAMPLE2	*SAMPLE2	1DDC	*SAMPLE2	*SAMPLE2
1DE0	*SAMPLE2	*SAMPLE2	1E80	*SAMPLE2	*SAMPLE2
1E88	*SAMPLE2	*SAMPLE2	1E8C	*SAMPLE2	*SAMPLE2
1F24	*SAMPLE2	*SAMPLE2	1F2C	*SAMPLE2	*SAMPLE2
1F30	*SAMPLE2	*SAMPLE2	2060	*SAMPLE2	*SAMPLE2
2068	*SAMPLE2	*SAMPLE2	206C	*SAMPLE2	*SAMPLE2
2480	*SAMPLE2	*SAMPLE2	2488	*SAMPLE2	*SAMPLE2
248C	*SAMPLE2	*SAMPLE2	2640	CEEUOPT	CEEUOPT
2634	CEEBXITA	\$UNRESOLVED (w)	2638	CEEBINT	CEEBINT
263C	CEEBLLST	CEEBLLST	2644	CEEBTRM	CEEBTRM
26F4	CEEP#INT	CEEP#INT	275C	CEEP#INT	CEEP#INT
2798	CEEP#INT	CEEP#INT	2740	CEEP#CAL	CEEP#CAL
276C	CEEP#CAL	CEEP#CAL	277C	CEEP#TRM	CEEP#TRM
27F4	CEEP#TRM	CEEP#TRM	29F8	CEEARLU	CEEARLU

LOCATION	REFERS TO	SYMBOL	IN CONTROL	SECTION	LOCATION	REFERS TO	SYMBOL	IN CONTROL	SECTION
2A04		CEEINT		CEEBPIRA	29E8		CEEFMAIN		\$UNRESOLVED (w)
29EC		CEEMAIN		CEEMAIN	29F0		PLIMAIN		\$UNRESOLVED (w)
29F4		IBMSEMNA		\$UNRESOLVED (w)	29FC		CEESG010		CEESG010
2A00		CEEOPAPI		CEEOPAPI	2A08		CEEROOTB		\$UNRESOLVED (w)
2B1C		CEEBETBL		CEEBETBL	2B20		IBMSMATH		CEEPMATH
2AD4		CEEMAIN		CEEMAIN	2B10		CEEFMAIN		\$UNRESOLVED (w)
2B0C		PLISTART		\$UNRESOLVED (w)	2AF4		PLIXOPT		PLIXOPT
2AF8		IBMBPOPT		\$UNRESOLVED (w)	2AD8		SYSPINT		SYSPINT
2AE0		PLITABS		\$UNRESOLVED (w)	2B00		IBMBEATA		IBMSEATA
2AD0		PLIMAIN		\$UNRESOLVED (w)	2B38		CEESTART		CEESTART
2B3C		CEEBETBL		CEEBETBL	2B28		CEEMAIN		CEEMAIN
2B34		CEEMAIN		CEEMAIN	2E10		CEESG000		\$UNRESOLVED (w)
2E14		CEESG001		\$UNRESOLVED (w)	2E18		CEESG002		\$UNRESOLVED (w)
2E1C		CEESG003		\$UNRESOLVED (w)	2E20		CEESG004		\$UNRESOLVED (w)
2E24		CEESG005		\$UNRESOLVED (w)	2E28		CEESG006		\$UNRESOLVED (w)
2E2C		CEESG007		\$UNRESOLVED (w)	2E30		CEESG008		\$UNRESOLVED (w)
2E34		CEESG009		\$UNRESOLVED (w)	2E38		CEESG010		CEESG010
2E3C		CEESG011		\$UNRESOLVED (w)	2E40		CEESG012		\$UNRESOLVED (w)
2E44		CEESG013		\$UNRESOLVED (w)	2E48		CEESG014		\$UNRESOLVED (w)
2E4C		CEESG015		\$UNRESOLVED (w)	2E50		CEESG016		\$UNRESOLVED (w)

```

LOCATION      3C REQUESTS CUMULATIVE PSEUDO REGISTER LENGTH
PSEUDO REGISTERS
NAME ORIGIN LENGTH      NAME ORIGIN LENGTH      NAME ORIGIN LENGTH
SOURCE      00      4
TOTAL LENGTH OF PSEUDO REGISTERS      4
ENTRY ADDRESS      00
TOTAL LENGTH      3C78
** GO          DID NOT PREVIOUSLY EXIST BUT WAS ADDED AND HAS AMODE 31
** LOAD MODULE HAS RMODE ANY
** AUTHORIZATION CODE IS      0.

```

```

*****
*** Word-use Report *** 1
*****

```

```

-count-  --word--
  3     BEGIN
  1     CLOSE
 13     DCL
 24     DECLARE
  2     DISPLAY
 14     DO
 13     ELSE
 23     END
  1     GO
 13     IF

```

Sample program output.

**1** Program output header

**2** The apparent error is intentional

```

-----The previous value should have been      14 2
  7     LIST
  4     ON
  1     OPEN
  2     PROC
  3     PROCEDURE
  2     READ
  4     RETURN
  1     SELECT
  2     STOP
 13     THEN
  2     WHEN

```

There were 148 references to 36 words.  
There was a discrepancy in at least one of the word-counts. **2**





## STORAGE REQUIREMENTS

<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>			
BLOCK, SECTION OR STATEMENT	TYPE	LENGTH (DEC)	LENGTH (HEX)	DSA SIZE (DEC)	DSA SIZE (HEX)	
*SAMPLE1	PROGRAM CSECT	4444	115C			
*SAMPLE2	STATIC CSECT	2880	B40			
SAMPLE	PROCEDURE BLOCK	2298	8FA	1208	4B8	
BLOCK 2 STMT 22	ON UNIT	170	AA	208	D0	
BLOCK 3 STMT 26	ON UNIT	162	A2	216	D8	
BLOCK 4 STMT 29	ON UNIT	304	130	288	120	
NEXT_WORD	PROCEDURE BLOCK	1056	420	368	170	
LOOKUP_WORD	PROCEDURE BLOCK	448	1C0	256	100	

Storage requirements. This table gives the main storage requirements for the program. These quantities do not include the main storage required by the library subroutines that will be included by the linkage editor or loaded dynamically during execution.

- 5** Name of the block, section, or number of the statement in the program.
- 6** Description of the block, section, or statement.
- 7** Length in bytes of the storage areas in both decimal and hexadecimal notation.
- 8** Length in bytes of the dynamic storage area (DSA) in both decimal and hexadecimal notation.

EXTERNAL SYMBOL DICTIONARY

<b>1</b> SYMBOL	<b>2</b> TYPE	<b>3</b> ID	<b>4</b> ADDR	<b>5</b> LENGTH
CEESTART	SD	0001	000000	000080
*SAMPLE1	SD	0002	000000	00115C
*SAMPLE2	SD	0003	000000	000B40
CEEMAIN	WX	0004	000000	
CEEMAIN	SD	0005	000000	000010
IBMRINP1	ER	0006	000000	
CEEFMAIN	WX	0007	000000	
CEEBETBL	ER	0008	000000	
CEERootA	ER	0009	000000	
CEEOP1PI	ER	000A	000000	
CEESG010	ER	000B	000000	
IBMSEATA	ER	000C	000000	
IELCGOG	SD	000D	000000	0000AE
IELCGOH	SD	000E	000000	0000A0
IELCGOC	SD	000F	000000	00007C
IELCGMY	SD	0010	000000	0000A4
IELCGCY	SD	0011	000000	00007E
IBMSIOA	ER	0012	000000	
IBMSASCA	ER	0013	000000	
IBMSCEDB	ER	0014	000000	
IBMSCHFD	ER	0015	000000	
IBMSCHXH	WX	0016	000000	
IBMSCWDH	ER	0017	000000	
IBMSEOCA	ER	0018	000000	
IBMSJDSA	ER	0019	000000	
IBMSOCLA	ER	001A	000000	
IBMSOCLC	WX	001B	000000	
IBMSRIOA	ER	001C	000000	
IBMSSEOA	ER	001D	000000	
IBMSIOE	WX	001E	000000	
IBMSIOT	WX	001F	000000	
IBMSLLOA	ER	0020	000000	
IBMSPLA	ER	0021	000000	
IBMSXCA	ER	0022	000000	
IBMSXCB	WX	0023	000000	
IBMSIST	WX	0024	000000	
CEEUOPT	SD	0025	000000	0004D0
PLIXOPT	SD	0026	000000	000066
PLIXOPT*	SD	0027	000000	00002C
PLIXOPT+	LD		000004	
PLIXOPT-	LD		000020	
PLIXOPT-	ER	0028	000000	
SAMPLE	LD		000008	
SAMPLE	ER	0029	000000	
SAMPLE*	SD	002A	000000	000020
SAMPLE+	LD		000000	
SOURCE	SD	002B	000000	00001C

External symbol dictionary

- 1** List of all the external symbols that make up the object module.
- 2** Type of external symbol, as follows:  
 CM Common area  
 ER External reference  
 LD Label definition  
 PR Pseudo-register  
 SD Section definition  
 WX Weak external reference  
 Full definitions of all these terms are given in "External symbol dictionary" in the main text.
- 3** All entries, except LD type entries, are identified by a hexadecimal number.
- 4** Address (in hexadecimal) of LD type entries.
- 5** Length in bytes (in hexadecimal) of SD, CM, and PR type entries.

SOURCE	PR	002C	000000	000004
SOURCE*	SD	002D	000000	000020
SOURCE+	LD		000000	
SYSPIINT	SD	002E	000000	000020
SYSPIINT*	SD	002F	000000	000020
SYSPIINT+	LD		000000	

<b>1</b>	<b>2</b>	<b>3</b>	<b>1</b>	<b>2</b>	<b>3</b>
		STATIC INTERNAL STORAGE MAP	0000C8	60000006	FED
			0000CC	58000009	FED
			0000D0	5800000C	FED
			0000D4	58000023	FED
			0000D8	2800	DED..WORD
			0000DA	2401	DED..FALSE
			0000DC	0002	CONSTANT
			0000DE	0048	CONSTANT
			0000E0	0000	CONSTANT
			0000E2	0001	CONSTANT
			0000E4	0003	CONSTANT
			0000E6	0005	CONSTANT
			0000E8	000A	CONSTANT
			0000EA	000D	CONSTANT
			0000EC	000E	CONSTANT
			0000EE	0012	CONSTANT
			0000F0	0013	CONSTANT
			0000F2	0016	CONSTANT
			0000F4	0018	CONSTANT
			0000F6	001A	CONSTANT
			0000F8	001E	CONSTANT
			0000FA	0021	CONSTANT
			0000FC	0022	CONSTANT
			0000FE	0017	CONSTANT
			000100	0007	CONSTANT
			000102	0004	CONSTANT
			000104	0024	CONSTANT
			000106	0009	CONSTANT
			000108	4040402020202020 202020202120	CONSTANT
			000116	001C	CONSTANT
			000118	001D	CONSTANT
			00011A		
			000120	00000000000001B4	LOCATOR..CONTROLLED_SET
			000128	00000000000001C4	LOCATOR..WORD_INDEX_TABLE
			000130	00000000000001D4	LOCATOR..WORD_TABLE
			000138	00000000001F8000	LOCATOR..WORD
			000140	00000535001E0000	LOCATOR..WORD_NEXT_CHARACTERS
			000148	00000518001D0000	LOCATOR..WORD_FIRST_CHARACTERS
			000150	0000000000010000	LOCATOR..FALSE
			000158	0000000000798000	LOCATOR..RECORD
			000160	0000000000008000	LOCATOR..DATA_RECORD
			000168	0080000091102000	CONSTANT
			000170	000000000200007B	RECORD_DESCRIPTOR
			000178	000003D800190000	LOCATOR
			000180	000003F100190000	LOCATOR
			000188	0000040A00140000	LOCATOR
			000190	0000000000340000	LOCATOR
			000198	00000000003B0000	LOCATOR
			0001A0	000004AE00280000	LOCATOR
			0001A8	0000000002D0000	LOCATOR
			0001B0	91E091E0	CONSTANT

Static internal storage map.  
This is a storage map of the static control section for the program. This control section is the third standard entry in the external symbol dictionary.

- 1** Six-digit offset (in hexadecimal)
- 2** Text (in hexadecimal)
- 3** Comment indicating type of item to which the text refers. A comment appears only against the first line of the text for an item.

```

5688-235 IBM PL/I for MVS & VM          /* PL/I Sample Program: Used to verify product installation */PAGE 37
0001B4 0000000200000002  DESCRIPTOR          40
0000002400000001          0002A6 C3C1D3D340404040  CONSTANT
0001C4 0000000200000002  DESCRIPTOR          40
0000001A00000001          0002AF C3D3D6E2C5404040  CONSTANT
0001D4 0000000900000009  DESCRIPTOR          40
0000002500000001          0002B8 C4C3D34040404040  CONSTANT
00090000          40
0001E8 00000001          CONSTANT          0002C1 C4C5C3D3C1D9C540  CONSTANT
0001EC 0010000000601800  CONSTANT          40
00000000          0002CA C4C5C6C1E4D3E340  CONSTANT
0001F8 00000002          CONSTANT          40
0001FC 00000003          CONSTANT          0002D3 C4C9E2D7D3C1E840  CONSTANT
000200 0000001F          CONSTANT          40
000204 00000000          0002DC C4D6404040404040  CONSTANT
000208 00000000          A..PLIXOPT          40
00020C 00000000          A..DCLCB            40
000210 000001E8          A..CONSTANT        0002E5 C5D3E2C540404040  CONSTANT
000214 00000000          A..DCLCB            0002EE C5D5C44040404040  CONSTANT
000218 000001EC          A..CONSTANT        40
00021C 00000000          OMITTED ARGUMENT   0002F7 C5D5E3D9E8404040  CONSTANT
000220 00000000          OMITTED ARGUMENT   40
000224 80000000          OMITTED ARGUMENT   000300 C6D9C5C540404040  CONSTANT
000228 00000000          A..DCLCB            40
00022C 00000168          A..CONSTANT        000309 C7C5D5C5D9C9C340  CONSTANT
000230 00000000          A..RD              40
000234 00000000          OMITTED ARGUMENT   000312 C7C5E34040404040  CONSTANT
000238 00000000          OMITTED ARGUMENT   40
00023C 80000000          OMITTED ARGUMENT   00031B C7D6404040404040  CONSTANT
000240 00000000          A..LOCATOR         40
000244 80000000          A..TEMP            000324 C7D6E3D640404040  CONSTANT
000248 00000000          A..LOCATOR         40
00024C 80000000          A..WORD_INDEX      00032D C9C6404040404040  CONSTANT
000250 000001E8          A..CONSTANT        40
000254 00000000          A..DCLCB            000336 D3C5C1E5C5404040  CONSTANT
000258 80000000          OMITTED ARGUMENT   40
00025C 00000000          A..DCLCB            00033F D3C9E2E340404040  CONSTANT
000260 00000000          A..TEMP            40
000264 800001E8          A..CONSTANT        000348 D3D6C3C1E3C54040  CONSTANT
000268 00000000          A..DCLCB            40
00026C 00000000          A..TEMP            000351 D6D5404040404040  CONSTANT
000270 800001F8          A..CONSTANT        40
000274 00000000          A..LOCATOR         00035A D6D7C5D540404040  CONSTANT
000278 000000E2          A..CONSTANT        40
00027C 000000BE          A..DED..WORD_COUNT 000363 D7D9D6C340404040  CONSTANT
000280 00000000          A..TEMP            40
000284 800000E0          A..CONSTANT        00036C D7D9D6C3C5C4E4D9  CONSTANT
000288 800001A0          A..CONSTANT        C5
00028C 0D800000          CONSTANT          000375 D9C5C1C440404040  CONSTANT
000290 80000000          A..TEMP            40
000294 C1D3D3D6C3C1E3C5    CONSTANT          00037E D9C5E3E4D9D54040  CONSTANT
40
00029D C2C5C7C9D5404040    CONSTANT          000387 D9C5E5C5D9E34040  CONSTANT

```

```

5688-235 IBM PL/I for MVS & VM          /* PL/I Sample Program: Used to verify product installation */PAGE 38
      40                                96A3408285859540
000390 D9C5E6D9C9E3C540      CONSTANT      848586899585844B
      40                                0004D6 4040202020202021      CONSTANT
000399 E2C5D3C5C3E34040      CONSTANT      20
      40                                0004DF E495A29785838986      CONSTANT
0003A2 E2C9C7D5C1D34040      CONSTANT      8985844085999996
      40                                9940968383A49999
0003AB E2E3D6D740404040      CONSTANT      85844B4040D6D5C3
      40                                D6C4C57E
0003B4 E3C8C5D540404040      CONSTANT      000503 615C      CONSTANT
      40                                000505 5C61      CONSTANT
0003BD E6C1C9E340404040      CONSTANT      000507 7D      CONSTANT
      40                                000508 0C16000000000A4C      STATIC ONCB
0003C6 E6C8C5D540404040      CONSTANT      000510 0C96000000000000      STATIC ONCB
      40                                000518 C1C2C3C4C5C6C7C8      INITIAL VALUE..WORD_FIRST_CHAR
0003CF E6D9C9E3C5404040      CONSTANT      C9D1D2D3D4D5D6D7
      40                                D8D9E2E3E4E5E6E7
0003D8 405C5C5C5C5C5C5C      CONSTANT      E8E97C7B5B
      5C5C5C5C5C5C5C5C      000535 C1C2C3C4C5C6C7C8      INITIAL VALUE..WORD_NEXT_CHAR
      5C5C5C5C5C5C5C5C      C9D1D2D3D4D5D6D7
      40                                D8D9E2E3E4E5E6E7
0003F1 405C5C5C40E69699      CONSTANT      E8E96D7C7B5B
      8460A4A28540D985      000558 00000000      SYMBOL TABLE ELEMENT
      979699A3405C5C5C      00055C 00000000      CONSTANT
      40                                000560 00000000      SYMBOL TABLE ELEMENT
00040A 40608396A495A360      CONSTANT      000564 00000604      SYMBOL TABLE ELEMENT
      4040406060A69699      000568 00000624      SYMBOL TABLE ELEMENT
      84606040      00056C 00000648      SYMBOL TABLE ELEMENT
00041E 6060606060606060      CONSTANT      000570 00000664      SYMBOL TABLE ELEMENT
      60606060      000574 80000000      SYMBOL TABLE ELEMENT
00042A E3888540979985A5      CONSTANT      000578 00000680      SYMBOL TABLE ELEMENT
      8996A4A240A58193      00057C 0000069C      SYMBOL TABLE ELEMENT
      A48540A28896A493      000580 000006B4      SYMBOL TABLE ELEMENT
      84408881A5854082      000584 000006DC      SYMBOL TABLE ELEMENT
      858595      000588 00000704      SYMBOL TABLE ELEMENT
00044D E38885998540A685      CONSTANT      00058C 0000071C      SYMBOL TABLE ELEMENT
      998540      000590 00000734      SYMBOL TABLE ELEMENT
000458 4099858685998595      CONSTANT      000594 00000754      SYMBOL TABLE ELEMENT
      8385A240A39640      000598 00000774      SYMBOL TABLE ELEMENT
000467 404040F3F6      CONSTANT      00059C 0000079C      SYMBOL TABLE ELEMENT
00046C 40A6969984A24B      CONSTANT      0005A0 000007BC      SYMBOL TABLE ELEMENT
000473 E38885998540A681      CONSTANT      0005A4 000007DC      SYMBOL TABLE ELEMENT
      A24081408489A283      0005A8 000008D0      SYMBOL TABLE ELEMENT
      998597819583A840      0005AC 000008EC      SYMBOL TABLE ELEMENT
      89954081A3409385      0005B0 00000000      SYMBOL TABLE ELEMENT
      81A2A34096958540      0005B4 00000000      SYMBOL TABLE ELEMENT
      968640A3888540A6      0005B8 00000000      CONSTANT
      969984608396A495      0005BC 00000558      SYMBOL TABLE ELEMENT
      A3A24B      0005C0 00000000      CONSTANT
0004AE E3888540899597A4      CONSTANT      0005C4 00000564      SYMBOL TABLE ELEMENT
      A3408481A38140A2      0005C8 00000000      CONSTANT
      85A3408881A24095      0005CC 00000564      SYMBOL TABLE ELEMENT

```

```

5688-235 IBM PL/I for MVS & VM          /* PL/I Sample Program: Used to verify product installation          */PAGE 39
0005D0 00000000 CONSTANT 0004E3D9E4C50000
0005D4 00000564 SYMBOL TABLE ELEMENT 000734 85000001000000BE SYMBOL TABLE..RIGHT_MARGIN
0005D8 000007F4 SYMBOL TABLE ELEMENT 0000011A00000000
0005DC 00000814 SYMBOL TABLE ELEMENT 000CD9C9C7C8E36D
0005E0 00000838 SYMBOL TABLE ELEMENT D4C1D9C7C9D50000
0005E4 00000858 SYMBOL TABLE ELEMENT 000754 85000001000000BE SYMBOL TABLE..LEFT_MARGIN
0005E8 00000874 SYMBOL TABLE ELEMENT 0000011C00000000
0005EC 00000000 CONSTANT 000BD3C5C6E36DD4
0005F0 00000564 SYMBOL TABLE ELEMENT C1D9C7C9D5000000
0005F4 00000894 SYMBOL TABLE ELEMENT 000774 81000001000000DA SYMBOL TABLE..DISCREPANCY_OCCU
0005F8 000008B4 SYMBOL TABLE ELEMENT 000000F800000000
0005FC 00000000 CONSTANT 0014C4C9E2C3D9C5
000600 00000564 SYMBOL TABLE ELEMENT D7C1D5C3E86DD6C3
000604 81000101000000BE SYMBOL TABLE..CONTROLLED_SET C3E4D9D9C5C40000
000000C000000000 00079C 85000001000000BE SYMBOL TABLE..LAST_CHAR_POSN
000EC3D6D5E3D9D6 0000011E00000000
D3D3C5C46DE2C5E3 000ED3C1E2E36DC3
000624 81000101000000BE SYMBOL TABLE..WORD_INDEX_TABLE C8C1D96DD7D6E2D5
000000C800000000 0007BC 81000001000000DA SYMBOL TABLE..RECORD_READ
0010E6D6D9C46DC9 0000010000000000
D5C4C5E76DE3C1C2 000BD9C5C3D6D9C4
D3C50000 6DD9C5C1C4000000
000648 81000101000000BE SYMBOL TABLE..WORD_COUNT 0007DC 81000001000000D8 SYMBOL TABLE..RECORD
000000D000000000 0000010800000000
000AE6D6D9C46DC3 0006D9C5C3D6D9C4
D6E4D5E3 0007F4 85000002000000BE SYMBOL TABLE..NEXT_CHAR_POSN
000664 81000101000000BC SYMBOL TABLE..WORD_TABLE 000000D000000000
000000D800000000 000ED5C5E7E36DC3
000AE6D6D9C46DE3 C8C1D96DD7D6E2D5
C1C2D3C5 000814 85000002000000BE SYMBOL TABLE..LENGTH_OF_STRING
000680 85000001000000BE SYMBOL TABLE..WORD_INDEX 000000D200000000
0000011800000000 0010D3C5D5C7E3C8
000AE6D6D9C46DC9 6DD6C66DE2E3D9C9
D5C4C5E7 D5C70000
00069C 81000001000000D8 SYMBOL TABLE..WORD 000838 81000002000000BC SYMBOL TABLE..NEXT_CHARACTER
000000E000000000 000000B800000000
0004E6D6D9C40000 000ED5C5E7E36DC3
0006B4 01000000000000BC SYMBOL TABLE..WORD_NEXT_CHARAC C8C1D9C1C3E3C5D9
0000014000000000 000858 81000002000000D8 SYMBOL TABLE..DATA_WORD
0014E6D6D9C46DD5 000000C000000000
C5E7E36DC3C8C1D9 0009C4C1E3C16DE6
C1C3E3C5D9E20000 D6D9C400
0006DC 01000000000000BC SYMBOL TABLE..WORD_FIRST_CHARA000874 A1000002000000D8 SYMBOL TABLE..DATA_RECORD
0000014800000000 0000010000000000
0015E6D6D9C46DC6 000BC4C1E3C16DD9
C9D9E2E36DC3C8C1 C5C3D6D9C4000000
D9C1C3E3C5D9E200 000894 85000002000000BE SYMBOL TABLE..WORD_NUMBER
000704 81000001000000DA SYMBOL TABLE..FALSE 000000C000000000
000000E800000000 000BE6D6D9C46DD5
0005C6C1D3E2C500 E4D4C2C5D9000000
00071C 81000001000000DA SYMBOL TABLE..TRUE 0008B4 A1000002000000D8 SYMBOL TABLE..DATA_WORD
000000F000000000 000000D000000000

```



WITHIN PROCEDURE SAMPLE															
OFFSET (HEX)	0	348	358	368	370	37E	388	39E	3AE	3BA	410	432	462	48A	492
STATEMENT NO.	1	22	26	29	34	35	36	37	38	39	40	41	42	43	44
OFFSET (HEX)	4E8	4F0	506	50E	51C	55C	59C	5DC	61C	62E	63A	642	6DA	6E2	6FA
STATEMENT NO.	45	46	47	48	49	50	51	52	53	54	53	54	55	56	57
OFFSET (HEX)	78A	794	79C	7A4	7A8	7B4	7CC	7CC	86E	8D2	8DA				
STATEMENT NO.	58	59	60	61	54	61	53	62	63	64	104				
WITHIN ON UNIT BLOCK 2															
OFFSET (HEX)	0	76	84	92											
STATEMENT NO.	22	23	24	25											
WITHIN ON UNIT BLOCK 3															
OFFSET (HEX)	0	80	8A												
STATEMENT NO.	26	27	28												
WITHIN ON UNIT BLOCK 4															
OFFSET (HEX)	0	90	98	10A	118										
STATEMENT NO.	29	30	31	32	33										
WITHIN PROCEDURE NEXT_WORD															
OFFSET (HEX)	0	BC	C8	100	11A	124	134	1C0	1C8	1D0	24C	254	292	29A	2AC
STATEMENT NO.	65	71	72	73	74	75	76	77	78	79	80	81	82	83	84
OFFSET (HEX)	328	386	38A	38E	3A4	3B8	3BC	3C0	3C0	3C4	3DC	414			
STATEMENT NO.	85	86	84	86	87	88	89	90	90	91	92	93			
WITHIN PROCEDURE LOOKUP_WORD															
OFFSET (HEX)	0	98	DA	E2	132	13A	13E	146	148	15E	176	182	184	18C	190
STATEMENT NO.	94	97	98	99	100	101	98	101	98	101	98	101	98	101	102
OFFSET (HEX)	1B4														
STATEMENT NO.	103														





MVS/DFP VERSION 3 RELEASE 3 LINKAGE EDITOR 10:03:27 FRI JAN 29, 1993  
 JOB IEL111VP STEP IVP PROCEDURE LKED  
 INVOCATION PARAMETERS - XREF,LIST **1**  
 ACTUAL SIZE=(317440,79872)  
 OUTPUT DATA SET SYS93029.T100323.RA000.IEL111VP.GOSET IS ON VOLUME PUB002

<b>3</b>			<b>2</b> CROSS REFERENCE TABLE							
CONTROL	SECTION		ENTRY							
NAME	ORIGIN	LENGTH	NAME	LOCATION	NAME	LOCATION	NAME	LOCATION	NAME	LOCATION
CEESTART	00	80								
CEEMAIN	80	10								
SYSPINT	90	20								
SOURCE	B0	1C								
CEEUOPT	D0	4D0								
PLIXOPT	5A0	66								
PLIXOPT*	608	2C								
SAMPLE*	638	20	PLIXOPT+	60C	PLIXOPT-	628				
SOURCE*	658	20	SAMPLE+	638						
SYSPINT*	678	20	SOURCE+	658						
			SYSPINT+	678						

Linkage editor listing.

- 1** Statement identifying the version and level of the linkage editor and giving the options as specified in the PARM parameter of the EXEC statement that invokes the linkage editor
- 2** Cross reference table, consisting of a module map and the cross reference table
- 3** The module map shows each control section and its associated entry points, if any, listed across the page. An asterisk in column 9 after a name beginning with "IBM" indicates a library subroutine obtained by automatic library call.
- 4** The cross reference table gives all the locations in a control section at which a symbol is reference. UNRESOLVED(W) identifies a weak external reference that has not been resolved.

*SAMPLE2	698	B40		
IELCGOG	11D8	AE		
IELCGOH	1288	A0		
IELCGOC	1328	7C		
IELCGMY	13A8	A4		
IELCGCY	1450	7E		
*SAMPLE1	14D0	115C	SAMPLE	14D8
CEEBETBL*	2630	1C		
CEEOP1PI*	2650	208		
CEEROOTA*	2858	268		
CEESG010*	2AC0	64		
IBMRINP1*	2B28	24		
IBMSASCA*	2B50	14	IBMBASCA	2B50
IBMSCEDB*	2B68	14	IBMBCEDB	2B68
IBMSCHFD*	2B80	14	IBMBCHFD	2B80
IBMSEATA*	2B98	14		

IBMSSIOA*	2BB0	14	IBMBEATA	2B98						
IBMSCHXH*	2BC8	14	IBMBSIOA	2BB0						
IBMSCWDH*	2BE0	14	IBMBCHXH	2BC8						
			IBMBCWDH	2BE0						
NAME	ORIGIN	LENGTH	NAME	LOCATION	NAME	LOCATION	NAME	LOCATION	NAME	LOCATION
IBMSEOCA*	2BF8	14	IBMBEOCA	2BF8						
IBMSJDSA*	2C10	14	IBMBJDSA	2C10						
IBMSOCLA*	2C28	14	IBMBOCLA	2C28						
IBMSRIOA*	2C40	14	IBMBRIOA	2C40						
IBMSSEOA*	2C58	14	IBMBSEOA	2C58						
IBMSSIOE*	2C70	14	IBMBSIOE	2C70						
IBMSSIOT*	2C88	14	IBMBSIOT	2C88						
IBMSSLOA*	2CA0	14	IBMBSLOA	2CA0						
CEEARLU *	2CB8	140								
CEEBINT *	2DF8	8								
CEEBLLST*	2E00	5C	CEELLIST	2E10						
CEEBTRM *	2E60	180								
CEEP#CAL*	2FE0	120								
CEEP#INT*	3100	340								
CEEP#TRM*	3440	210								
IBMSOCLC*	3650	14	IBMBOCLC	3650						
IBMSSPLA*	3668	14	IBMBSPLA	3668						
IBMSSXCA*	3680	14	IBMBSXCA	3680						
CEEBPIRA*	3698	2E0	CEEINT	3698	CEEBPIRB	3698	CEEBPIRC	3698		
IBMSCEDF*	3978	14	IBMBCEDF	3978						
IBMSCEDX*	3990	14	IBMBCEDX	3990						
IBMSCEFX*	39A8	14	IBMBCEFX	39A8						
IBMSCEZB*	39C0	14	IBMBCEZB	39C0						
IBMSCEZF*	39D8	14	IBMBCEZF	39D8						
IBMSCEZX*	39F0	14	IBMBCEZX	39F0						
IBMSCHFE*	3A08	14	IBMBCHFE	3A08						
CEEPMATH*	3A20	18	IBMSMATH	3A20						
IBMSSXCB*	3A38	14	IBMBSXCB	3A38						
IBMSCHF*	3A50	14	IBMBCHF*	3A50						

NAME	ORIGIN	LENGTH	NAME	LOCATION	NAME	LOCATION	NAME	LOCATION	NAME	LOCATION
IBMSCHFP*	3A68	14	IBMBCHFP	3A68						
IBMSCHFY*	3A80	14	IBMBCHFY	3A80						
IBMSCHXD*	3A98	14	IBMBCHXD	3A98						
IBMSCHXE*	3AB0	14	IBMBCHXE	3AB0						
IBMSCHXF*	3AC8	14	IBMBCHXF	3AC8						
IBMSCHXP*	3AE0	14	IBMBCHXP	3AE0						
IBMSCHXY*	3AF8	14	IBMBCHXY	3AF8						
IBMSSIOB*	3B10	14	IBMBSIOB	3B10						
IBMSSIOC*	3B28	14	IBMBSIOC	3B28						
IBMSCWZH*	3B40	14	IBMBCWZH	3B40						
IBMSJDSB*	3B58	14	IBMBJDSB	3B58						
IBMSOCLB*	3B70	14	IBMBOCLB	3B70						
IBMSOCLD*	3B88	14	IBMBOCLD	3B88						
IBMSRIOB*	3BA0	14	IBMBRIOB	3BA0						
IBMSRIOC*	3BB8	14	IBMBRIOC	3BB8						
IBMSRIOD*	3BD0	14	IBMBRIOD	3BD0						
IBMSSIOD*	3BE8	14	IBMBSIOD	3BE8						
IBMSSLOB*	3C00	14	IBMBSLOB	3C00						
IBMSSPLB*	3C18	14	IBMBSPLB	3C18						
IBMSSPLC*	3C30	14	IBMBSPLC	3C30						
IBMSSXCC*	3C48	14	IBMBSXCC	3C48						
IBMSSXCD*	3C60	14	IBMBSXCD	3C60						

LOCATION	REFERS TO	SYMBOL	IN CONTROL SECTION
2C		CEEMAIN	CEEMAIN
74		CEEBETBL	CEEBETBL
84		*SAMPLE1	*SAMPLE1
62C		PLIXOPT	PLIXOPT

LOCATION	REFERS TO	SYMBOL	IN CONTROL SECTION
68		CEEFMAIN	4 \$UNRESOLVED(W)
78		CEEROTA	CEEROTA
88		IBMRINP1	IBMRINP1
640		SAMPLE	*SAMPLE1

LOCATION	REFERS TO	SYMBOL	IN CONTROL SECTION
660		SOURCE	SOURCE
69C		*SAMPLE1	*SAMPLE1
6A4		*SAMPLE1	*SAMPLE1
6AC		*SAMPLE1	*SAMPLE1
6B4		*SAMPLE1	*SAMPLE1
6BC		*SAMPLE1	*SAMPLE1
6C4		*SAMPLE1	*SAMPLE1
6CC		*SAMPLE1	*SAMPLE1
6D4		*SAMPLE1	*SAMPLE1
6DC		*SAMPLE1	*SAMPLE1
6E4		*SAMPLE1	*SAMPLE1

LOCATION	REFERS TO	SYMBOL	IN CONTROL SECTION
680		SYSPINT	SYSPINT
6A0		*SAMPLE1	*SAMPLE1
6A8		*SAMPLE1	*SAMPLE1
6B0		*SAMPLE1	*SAMPLE1
6B8		*SAMPLE1	*SAMPLE1
6C0		*SAMPLE1	*SAMPLE1
6C8		*SAMPLE1	*SAMPLE1
6D0		*SAMPLE1	*SAMPLE1
6D8		*SAMPLE1	*SAMPLE1
6E0		*SAMPLE1	*SAMPLE1
6E8		*SAMPLE1	*SAMPLE1

6EC	*SAMPLE1	*SAMPLE1	6F0	*SAMPLE1	*SAMPLE1
6F4	IELCGOG	IELCGOG	6F8	IELCGOH	IELCGOH
6FC	IELCGOC	IELCGOC	700	IELCGMY	IELCGMY
704	IELCGCY	IELCGCY	708	IBMSASCA	IBMSASCA
70C	IBMSCEDB	IBMSCEDB	710	IBMSCHFD	IBMSCHFD
714	IBMSCHXH	IBMSCHXH	718	IBMSCWDH	IBMSCWDH
71C	IBMSEOCA	IBMSEOCA	720	IBMSJDSA	IBMSJDSA
724	IBMSOCLA	IBMSOCLA	728	IBMSOCLC	IBMSOCLC
72C	IBMSRIOA	IBMSRIOA	730	IBMSSEOA	IBMSSEOA
734	IBMSSIOE	IBMSSIOE	738	IBMSSIOT	IBMSSIOT
73C	IBMSSLOA	IBMSSLOA	740	IBMSSPLA	IBMSSPLA
744	IBMSSXCA	IBMSSXCA	748	IBMSSXCB	IBMSSXCB
89C	PLIXOPT	PLIXOPT	8A0	SYSPINT	SYSPINT
8A4	SOURCE	SOURCE	8AC	SOURCE	SOURCE
8C0	SOURCE	SOURCE	8EC	SOURCE	SOURCE
8F4	SYSPINT	SYSPINT	900	SYSPINT	SYSPINT
BA4	*SAMPLE1	*SAMPLE1	BF0	SAMPLE*	SAMPLE*
C0C	PLIXOPT*	PLIXOPT*	C48	SOURCE*	SOURCE*
C4C	SYSPINT*	SYSPINT*	F70	*SAMPLE1	*SAMPLE1
F8C	*SAMPLE1	*SAMPLE1	FA8	*SAMPLE1	*SAMPLE1
106C	*SAMPLE1	*SAMPLE1	108C	*SAMPLE1	*SAMPLE1
10A8	*SAMPLE1	*SAMPLE1	10CC	*SAMPLE1	*SAMPLE1
1164	*SAMPLE1	*SAMPLE1	11D4	CEESTART	CEESTART
1320	IBMSSIST	\$UNRESOLVED (w)	1324	IBMSSEOA	IBMSSEOA
139C	IBMSSXCB	IBMSSXCB	13A0	IBMSSIST	\$UNRESOLVED (w)
14E0	*SAMPLE2	*SAMPLE2	14E8	*SAMPLE2	*SAMPLE2
14EC	*SAMPLE2	*SAMPLE2	1510	*SAMPLE2	*SAMPLE2
1DD4	*SAMPLE2	*SAMPLE2	1DDC	*SAMPLE2	*SAMPLE2
1DE0	*SAMPLE2	*SAMPLE2	1E80	*SAMPLE2	*SAMPLE2
1E88	*SAMPLE2	*SAMPLE2	1E8C	*SAMPLE2	*SAMPLE2
1F24	*SAMPLE2	*SAMPLE2	1F2C	*SAMPLE2	*SAMPLE2
1F30	*SAMPLE2	*SAMPLE2	2060	*SAMPLE2	*SAMPLE2
2068	*SAMPLE2	*SAMPLE2	206C	*SAMPLE2	*SAMPLE2
2480	*SAMPLE2	*SAMPLE2	2488	*SAMPLE2	*SAMPLE2
248C	*SAMPLE2	*SAMPLE2	2640	CEEUOPT	CEEUOPT
2634	CEEBXITA	\$UNRESOLVED (w)	2638	CEEBINT	CEEBINT
263C	CEEBLLST	CEEBLLST	2644	CEEBTRM	CEEBTRM
26F4	CEEP#INT	CEEP#INT	275C	CEEP#INT	CEEP#INT
2798	CEEP#INT	CEEP#INT	2740	CEEP#CAL	CEEP#CAL
276C	CEEP#CAL	CEEP#CAL	277C	CEEP#TRM	CEEP#TRM
27F4	CEEP#TRM	CEEP#TRM	29F8	CEEARLU	CEEARLU

LOCATION	REFERS TO	SYMBOL	IN CONTROL	SECTION	LOCATION	REFERS TO	SYMBOL	IN CONTROL	SECTION
2A04		CEEINT		CEEBPIRA	29E8		CEEFMAIN		\$UNRESOLVED (w)
29EC		CEEMAIN		CEEMAIN	29F0		PLIMAIN		\$UNRESOLVED (w)
29F4		IBMSEMNA		\$UNRESOLVED (w)	29FC		CEESG010		CEESG010
2A00		CEEOPAPI		CEEOPAPI	2A08		CEEROOTB		\$UNRESOLVED (w)
2B1C		CEEBETBL		CEEBETBL	2B20		IBMSMATH		CEEPMATH
2AD4		CEEMAIN		CEEMAIN	2B10		CEEFMAIN		\$UNRESOLVED (w)
2B0C		PLISTART		\$UNRESOLVED (w)	2AF4		PLIXOPT		PLIXOPT
2AF8		IBMBPOPT		\$UNRESOLVED (w)	2AD8		SYSPINT		SYSPINT
2AE0		PLITABS		\$UNRESOLVED (w)	2B00		IBMBEATA		IBMSEATA
2AD0		PLIMAIN		\$UNRESOLVED (w)	2B38		CEESTART		CEESTART
2B3C		CEEBETBL		CEEBETBL	2B28		CEEMAIN		CEEMAIN
2B34		CEEMAIN		CEEMAIN	2E10		CEESG000		\$UNRESOLVED (w)
2E14		CEESG001		\$UNRESOLVED (w)	2E18		CEESG002		\$UNRESOLVED (w)
2E1C		CEESG003		\$UNRESOLVED (w)	2E20		CEESG004		\$UNRESOLVED (w)
2E24		CEESG005		\$UNRESOLVED (w)	2E28		CEESG006		\$UNRESOLVED (w)
2E2C		CEESG007		\$UNRESOLVED (w)	2E30		CEESG008		\$UNRESOLVED (w)
2E34		CEESG009		\$UNRESOLVED (w)	2E38		CEESG010		CEESG010
2E3C		CEESG011		\$UNRESOLVED (w)	2E40		CEESG012		\$UNRESOLVED (w)
2E44		CEESG013		\$UNRESOLVED (w)	2E48		CEESG014		\$UNRESOLVED (w)
2E4C		CEESG015		\$UNRESOLVED (w)	2E50		CEESG016		\$UNRESOLVED (w)

```

LOCATION      3C REQUESTS CUMULATIVE PSEUDO REGISTER LENGTH
PSEUDO REGISTERS
  NAME  ORIGIN  LENGTH      NAME  ORIGIN  LENGTH      NAME  ORIGIN  LENGTH      NAME  ORIGIN  LENGTH
SOURCE    00    4
TOTAL LENGTH OF PSEUDO REGISTERS      4
ENTRY ADDRESS    00
TOTAL LENGTH    3C78
** GO          DID NOT PREVIOUSLY EXIST BUT WAS ADDED AND HAS AMODE 31
** LOAD MODULE HAS RMODE ANY
** AUTHORIZATION CODE IS          0.

```

```

*****
*** Word-use Report *** 1
*****

```

```

-count-  --word--
  3      BEGIN
  1      CLOSE
  13     DCL
  24     DECLARE
  2      DISPLAY
  14     DO
  13     ELSE
  23     END
  1      GO
  13     IF

```

Sample program output.

**1** Program output header

**2** The apparent error is intentional

```

-----The previous value should have been 14 2
  7      LIST
  4      ON
  1      OPEN
  2      PROC
  3      PROCEDURE
  2      READ
  4      RETURN
  1      SELECT
  2      STOP
  13     THEN
  2      WHEN

```

There were 148 references to 36 words.  
 There was a discrepancy in at least one of the word-counts. **2**

---

## Bibliography

---

### PL/I for MVS & VM Publications

- *Licensed Program Specifications*, GC26-3116
- *Installation and Customization under MVS*, SC26-3119
- *Compiler and Run-Time Migration Guide*, SC26-3118
- *Programming Guide*, SC26-3113
- *Language Reference*, SC26-3114
- *Reference Summary*, SX26-3821
- *Compile-Time Messages and Codes*, SC26-3229
- *Diagnosis Guide*, SC26-3149

---

### Language Environment for MVS & VM Publications

- *Fact Sheet*, GC26-4785
- *Concepts Guide*, GC26-4786
- *Licensed Program Specifications*, GC26-4774
- *Installation and Customization under MVS*, SC26-4817
- *Programming Guide*, SC26-4818
- *Programming Reference*, SC26-3312
- *Debugging Guide and Run-Time Messages*, SC26-4829
- *Writing Interlanguage Communication Applications*, SC26-8351
- *Run-Time Migration Guide*, SC26-8232
- *Master Index*, SC26-3427

---

### PL/I for OS/2 Publications

- *Programming Guide*, SC26-8001
- *Language Reference*, SC26-8003
- *Reference Summary*, SX26-3832
- *Built-In Functions*, SC26-8089
- *Installation*, SX26-3833
- *Messages and Codes*, SC26-8002
- *License Information*, GC26-8004
- *WorkFrame/2 Guide*, SC26-8000

---

### CoOperative Development Environment/370

- *Fact Sheet*, GC09-1861
- *General Information*, GC09-2048
- *Installation*, SC09-1624
- *Licensed Program Specifications*, GC09-1898
- *User's Guide and Reference*, SC09-1623
- *Using CODE/370 with VS COBOL II and OS PL/I*, SC09-1862
- *Self-Study Guide*, SC09-2047

---

### IBM Debug Tool

- *User's Guide and Reference*, SC09-2137

---

### Softcopy Publications

Online publications are distributed on CD-ROMs and can be ordered from Mechanicsburg through your IBM representative. PL/I books are distributed on the following collection kit:

- *Application Development Collection Kit*, SK2T-1237

---

### Other Books You Might Need

#### CICS/ESA

- *Application Programming Reference*, SC33-0676

#### DFSORT

- *Application Programming Guide*, SC33-4035

#### DFSORT/CMS

- *User's Guide*, SC26-4361

#### IMS

- *IMS/ESA V4 Application Programming: Database Manager*, SC26-3058
- *IMS/ESA V4 Application Programming: Design Guide*, SC26-3066
- *IMS/ESA V4 Application Programming: Transaction Manager*, SC26-3063
- *IMS/ESA V4 Application Programming: EXEC DLI Commands for CICS and IMS*, SC26-3062

#### MVS/DFP

- *Access Method Services*, SC26-4562

**MVS/ESA 4.3 MVS Support for OpenEdition Services Feature**

- *Introducing OpenEdition MVS*, GC23-3010
- *OpenEdition MVS POSIX.1 Conformance Document*, GC23-3011
- *OpenEdition MVS User's Guide*, SC23-3013
- *OpenEdition MVS Command Reference*, SC23-3014

**MVS/ESA**

- *JCL User's Guide*, GC28-1473
- *JCL Reference*, GC28-1479
- *System Generation*, CG28-1825
- *System Programming Library: Initialization and Tuning Guide*, GC28-1451
- *System Messages Volume 1*, GC28-1480
- *System Messages Volume 2*, GC28-1481
- *System Messages Volume 3*, GC28-1482
- *System Messages Volume 4*, GC28-1483
- *System Messages Volume 5*, GC28-1484

**OS/VS2**

- *TSO Command Language Reference*, GC28-0646
- *TSO Terminal User's Guide*, GC28-0645
- *Job Control Language*, GC28-0692
- *Message Library: VS2 System Codes*, GC38-1008

**SMP/E**

- *User's Guide*, SC28-1302
- *DBIPO Dialogs User's Guide*, SC23-0538
- *Reference*, SC28-1107
- *Reference Summary*, SX22-0006

**TCAM**

- *ACF TCAM Application Programmer's Guide*, SC30-3233
- *OS/VS TCAM Concepts and Applications*, GC30-2049

**TSO/E**

- *Command Reference*, SC28-1881

**VM/ESA**

- *CMS User's Guide*, SC24-5460
- *CMS Command Reference*, SC24-5461
- *CMS Application Development Guide*, SC24-5450
- *XEDIT User's Guide*, SC24-5463
- *XEDIT Command and Macro Reference*, SC24-5464
- *CP Command and Utility Reference*, SC24-5519
- *Installation*, SC24-5526
- *Service Guide*, SC24-5527
- *System Messages and Codes*, SC24-5529.

---

# Glossary

This glossary defines terms for all platforms and releases of PL/I. It might contain terms that this manual does not use. If you do not find the terms for which you are looking, see the index in this manual or *IBM Dictionary of Computing*, SC20-1699.

## A

**access.** To reference or retrieve data.

**action specification.** In an ON statement, the ON-unit or the single keyword SYSTEM, either of which specifies the action to be taken whenever the appropriate condition is raised.

**activate (a block).** To initiate the execution of a block. A procedure block is activated when it is invoked. A begin-block is activated when it is encountered in the normal flow of control, including a branch. A package cannot be activated.

**activate (a preprocessor variable or preprocessor entry point).** To make a macro facility identifier eligible for replacement in subsequent source code. The %ACTIVATE statement activates preprocessor variables or preprocessor entry points.

**active.** (1) The state of a block after activation and before termination. (2) The state in which a preprocessor variable or preprocessor entry name is said to be when its value can replace the corresponding identifier in source program text. (3) The state in which an event variable is said to be during the time it is associated with an asynchronous operation. (4) The state in which a task variable is said to be when its associated task is attached. (5) The state in which a task is said to be before it has been terminated.

**actual origin (AO).** The location of the first item in the array or structure.

**additive attribute.** A file description attribute for which there are no defaults, and which, if required, must be stated explicitly or implied by another explicitly stated attribute. Contrast with *alternative attribute*.

**adjustable extent.** The bound (of an array), the length (of a string), or the size (of an area) that might be different for different generations of the associated variable. Adjustable extents are specified as expressions or asterisks (or by REFER options for based variables), which are evaluated separately for each generation. They cannot be used for static variables.

**aggregate.** See *data aggregate*.

**aggregate expression.** An array, structure, or union expression.

**aggregate type.** For any item of data, the specification whether it is structure, union, or array.

**allocated variable.** A variable with which main storage is associated and not freed.

**allocation.** (1) The reservation of main storage for a variable. (2) A generation of an allocated variable. (3) The association of a PL/I file with a system data set, device, or file.

**alignment.** The storing of data items in relation to certain machine-dependent boundaries (for example, a fullword or halfword boundary).

**alphabetic character.** Any of the characters A through Z of the English alphabet and the alphabetic extenders #, \$, and @ (which can have a different graphic representation in different countries).

**alphanumeric character.** An alphabetic character or a digit.

**alternative attribute.** A file description attribute that is chosen from a group of attributes. If none is specified, a default is assumed. Contrast with *additive attribute*.

**ambiguous reference.** A reference that is not sufficiently qualified to identify one and only one name known at the point of reference.

**area.** A portion of storage within which based variables can be allocated.

**argument.** An expression in an argument list as part of an invocation of a subroutine or function.

**argument list.** A parenthesized list of zero or more arguments, separated by commas, following an entry name constant, an entry name variable, a generic name, or a built-in function name. The list becomes the parameter list of the entry point.

**arithmetic comparison.** A comparison of numeric values. See also *bit comparison*, *character comparison*.

**arithmetic constant.** A fixed-point constant or a floating-point constant. Although most arithmetic constants can be signed, the sign is not part of the constant.

**arithmetic conversion.** The transformation of a value from one arithmetic representation to another.

**arithmetic data.** Data that has the characteristics of base, scale, mode, and precision. Coded arithmetic data and pictured numeric character data are included.

**arithmetic operators.** Either of the prefix operators + and -, or any of the following infix operators: + - \* / \*\*

**array.** A named, ordered collection of one or more data elements with identical attributes, grouped into one or more dimensions.

**array expression.** An expression whose evaluation yields an array of values.

**array of structures.** An ordered collection of identical structures specified by giving the dimension attribute to a structure name.

**array variable.** A variable that represents an aggregate of data items that must have identical attributes. Contrast with *structure variable*.

**ASCII.** American National Standard Code for Information Interchange.

**assignment.** The process of giving a value to a variable.

**asynchronous operation.** (1) The overlap of an input/output operation with the execution of statements. (2) The concurrent execution of procedures using multiple flows of control for different tasks.

**attachment of a task.** The invocation of a procedure and the establishment of a separate flow of control to execute the invoked procedure (and procedures it invokes) asynchronously, with execution of the invoking procedure.

**attention.** An occurrence, external to a task, that could cause a task to be interrupted.

**attribute.** (1) A descriptive property associated with a name to describe a characteristic represented. (2) A descriptive property used to describe a characteristic of the result of evaluation of an expression.

**automatic storage allocation.** The allocation of storage for automatic variables.

**automatic variable.** A variable whose storage is allocated automatically at the activation of a block and released automatically at the termination of that block.

## B

**base.** The number system in which an arithmetic value is represented.

**base element.** A member of a structure or a union that is itself not another structure or union.

**base item.** The automatic, controlled, or static variable or the parameter upon which a defined variable is defined.

**based reference.** A reference that has the based storage class.

**based storage allocation.** The allocation of storage for based variables.

**based variable.** A variable whose storage address is provided by a locator. Multiple generations of the same variable are accessible. It does not identify a fixed location in storage.

**begin-block.** A collection of statements delimited by BEGIN and END statements, forming a name scope. A begin-block is activated either by the raising of a condition (if the begin-block is the action specification for an ON-unit) or through the normal flow of control, including any branch resulting from a GOTO statement.

**binary.** A number system whose only numerals are 0 and 1.

**binary digit.** See *bit*.

**binary fixed-point value.** An integer consisting of binary digits and having an optional binary point and optional sign. Contrast with *decimal fixed-point value*.

**binary floating-point value.** An approximation of a real number in the form of a significand, which can be considered as a binary fraction, and an exponent, which can be considered as an integer exponent to the base of 2. Contrast with *decimal floating-point value*.

**bit.** (1) A 0 or a 1. (2) The smallest amount of space of computer storage.

**bit comparison.** A left-to-right, bit-by-bit comparison of binary digits. See also *arithmetic comparison*, *character comparison*.

**bit string constant.** (1) A series of binary digits enclosed in and followed immediately by the suffix B. Contrast with *character constant*. (2) A series of hexadecimal digits enclosed in single quotes and followed by the suffix B4.

**bit string.** A string composed of zero or more bits.

**bit string operators.** The logical operators not and exclusive-or (¬), and (&), and or (|).

**bit value.** A value that represents a bit type.

**block.** A sequence of statements, processed as a unit, that specifies the scope of names and the allocation of storage for names declared within it. A block can be a package, procedure, or a begin-block.

**bounds.** The upper and lower limits of an array dimension.

**break character.** The underscore symbol (\_). It can be used to improve the readability of identifiers. For instance, a variable could be called OLD\_INVENTORY\_TOTAL instead of OLDINVENTORYTOTAL.

**built-in function.** A predefined function supplied by the language, such as SQRT (square root).

**built-in function reference.** A built-in function name, which has an optional argument list.

**built-in name.** The entry name of a built-in subroutine.

**built-in subroutine.** Subroutine that has an entry name that is defined at compile-time and is invoked by a CALL statement.

**buffer.** Intermediate storage, used in input/output operations, into which a record is read during input and from which a record is written during output.

## C

**call.** To invoke a subroutine by using the CALL statement or CALL option.

**character comparison.** A left-to-right, character-by-character comparison according to the collating sequence. See also *arithmetic comparison*, *bit comparison*.

**character string constant.** A sequence of characters enclosed in single quotes; for example, 'Shakespeare' 's "Hamlet" '.

**character set.** A defined collection of characters. See *language character set* and *data character set*. See also *ASCII* and *EBCDIC*.

**character string picture data.** Picture data that has only a character value. This type of picture data must have at least one A or X picture specification character. Contrast with *numeric picture data*.

**closing (of a file).** The dissociation of a file from a data set or device.

**coded arithmetic data.** Data items that represent numeric values and are characterized by their base (decimal or binary), scale (fixed-point or floating-point), and precision (the number of digits each can have). This data is stored in a form that is acceptable, without conversion, for arithmetic calculations.

**combined nesting depth.** The deepest level of nesting, determined by counting the levels of PROCEDURE/BEGIN/ON, DO, SELECT, and IF...THEN...ELSE nestings in the program.

**comment.** A string of zero or more characters used for documentation that are delimited by /\* and \*/.

**commercial character.**

- CR (credit) picture specification character
- DB (debit) picture specification character

**comparison operator.** An operator that can be used in an arithmetic, string locator, or logical relation to indicate the comparison to be done between the terms in the relation. The comparison operators are:

= (equal to)  
> (greater than)  
< (less than)  
>= (greater than or equal to)  
<= (less than or equal to)  
¬= (not equal to)  
¬> (not greater than)  
¬< (not less than).

**compile time.** In general, the time during which a source program is translated into an object module. In PL/I, it is the time during which a source program can be altered, if desired, and then translated into an object program.

**compiler options.** Keywords that are specified to control certain aspects of a compilation, such as: the nature of the object module generated, the types of printed output produced, and so forth.

**complex data.** Arithmetic data, each item of which consists of a real part and an imaginary part.

**composite operator.** An operator that consists of more than one special character, such as <=, \*\*, and /\*.

**compound statement.** A statement that contains other statements. In PL/I, IF, ON, OTHERWISE, and WHEN are the only compound statements. See *statement body*.

**concatenation.** The operation that joins two strings in the order specified, forming one string whose length is equal to the sum of the lengths of the two original strings. It is specified by the operator ||.

**condition.** An exceptional situation, either an error (such as an overflow), or an expected situation (such as the end of an input file). When a condition is raised (detected), the action established for it is processed. See also *established action* and *implicit action*.

**condition name.** Name of a PL/I-defined or programmer-defined condition.

**condition prefix.** A parenthesized list of one or more condition names prefixed to a statement. It specifies whether the named conditions are to be enabled or disabled.

**connected aggregate.** An array or structure whose elements occupy contiguous storage without any intervening data items. Contrast with *nonconnected aggregate*.

**connected reference.** A reference to connected storage. It must be apparent, prior to execution of the program, that the storage is connected.

**connected storage.** Main storage of an uninterrupted linear sequence of items that can be referred to by a single name.

**constant.** (1) An arithmetic or string data item that does not have a name and whose value cannot change.  
(2) An identifier declared with the VALUE attribute.  
(3) An identifier declared with the FILE or the ENTRY attribute but without the VARIABLE attribute.

**constant reference.** A value reference which has a constant as its object

**contained block, declaration, or source text.** All blocks, procedures, statements, declarations, or source text inside a begin, procedure, or a package block. The entire package, procedure, and the BEGIN statement and its corresponding END statements are not contained in the block.

**containing block.** The package, procedure, or begin-block that contains the declaration, statement, procedure, or other source text in question.

**contextual declaration.** The appearance of an identifier that has not been explicitly declared in a DECLARE statement, but whose context of use allows the association of specific attributes with the identifier.

**control character.** A character in a character set whose occurrence in a particular context specifies a control function. One example is the end-of-file (EOF) marker.

**control format item.** A specification used in edit-directed transmission to specify positioning of a data item within the stream or printed page.

**control variable.** A variable that is used to control the iterative execution of a DO statement.

**controlled parameter.** A parameter for which the CONTROLLED attribute is specified in a DECLARE statement. It can be associated only with arguments that have the CONTROLLED attribute.

**controlled storage allocation.** The allocation of storage for controlled variables.

**controlled variable.** A variable whose allocation and release are controlled by the ALLOCATE and FREE statements, with access to the current generation only.

**conversion.** The transformation of a value from one representation to another to conform to a given set of attributes. For example, converting a character string to an arithmetic value such as FIXED BINARY (15,0).

**cross section of an array.** The elements represented by the extent of at least one dimension of an array. An asterisk in the place of a subscript in an array reference indicates the entire extent of that dimension.

**current generation.** The generation of an automatic or controlled variable that is currently available by referring to the name of the variable.

## D

**DDM file.** A &system. file that is associated with a remote file that is accessed using DDM. The DDM file provides the information needed for a local (source) system to locate a remote (target) system and to access the file at the target system where the requested data is stored.

**data.** Representation of information or of value in a form suitable for processing.

**data aggregate.** A data item that is a collection of other data items.

**data attribute.** A keyword that specifies the type of data that the data item represents, such as FIXED BINARY.

**data-directed transmission.** The type of stream-oriented transmission in which data is transmitted. It resembles an assignment statement and is of the form:

name = constant

**data item.** A single named unit of data.

**data list.** In stream-oriented transmission, a parenthesized list of the data items used in GET and PUT statements. Contrast with *format list*.

**data set.** (1) A collection of data external to the program that can be accessed by reference to a single file name. (2) A device that can be referenced.

**data specification.** The portion of a stream-oriented transmission statement that specifies the mode of transmission (DATA, LIST, or EDIT) and includes the data list(s) and, for edit-directed mode, the format list(s).

**data stream.** Data being transferred from or to a data set by stream-oriented transmission, as a continuous stream of data elements in character form.

**data transmission.** The transfer of data from a data set to the program or vice versa.

**data type.** A set of data attributes.

**DBCS.** In the character set, each character is represented by two consecutive bytes.

**deactivated.** The state in which an identifier is said to be when its value cannot replace a preprocessor identifier in source program text. Contrast with *active*.

**debugging.** Process of removing bugs from a program.

**decimal.** The number system whose numerals are 0 through 9.

**decimal digit.** One of the digits 0 through 9.

**decimal digit picture character.** The picture specification character 9.

**decimal fixed-point constant.** A constant consisting of one or more decimal digits with an optional decimal point.

**decimal fixed-point value.** A rational number consisting of a sequence of decimal digits with an assumed position of the decimal point. Contrast with *binary fixed-point value*.

**decimal floating-point constant.** A value made up of a significand that consists of a decimal fixed-point constant, and an exponent that consists of the letter E followed by an optionally signed integer constant not exceeding three digits.

**decimal floating-point value.** An approximation of a real number, in the form of a significand, which can be considered as a decimal fraction, and an exponent, which can be considered as an integer exponent to the base of 10. Contrast with *binary floating-point value*.

**decimal picture data.** See *numeric picture data*.

**declaration.** (1) The establishment of an identifier as a name and the specification of a set of attributes

(partial or complete) for it. (2) A source of attributes of a particular name.

**default.** Describes a value, attribute, or option that is assumed when none has been specified.

**defined variable.** A variable that is associated with some or all of the storage of the designated base variable.

**delimit.** To enclose one or more items or statements with preceding and following characters or keywords.

**delimiter.** All comments and the following characters: percent, parentheses, comma, period, semicolon, colon, assignment symbol, blank, pointer, asterisk, and single quote. They define the limits of identifiers, constants, picture specifications, iSUBs, and keywords.

**descriptor.** A control block that holds information about a variable, such as area size, array bounds, or string length.

**digit.** One of the characters 0 through 9.

**dimension attribute.** An attribute that specifies the number of dimensions of an array and indicates the bounds of each dimension.

**disabled.** The state of a condition in which no interrupt occurs and no established action will take place.

**do-group.** A sequence of statements delimited by a DO statement and ended by its corresponding END statement, used for control purposes. Contrast with *block*.

**do-loop.** See *iterative do-group*.

**dummy argument.** Temporary storage that is created automatically to hold the value of an argument that cannot be passed by reference.

**dump.** Printout of all or part of the storage used by a program as well as other program information, such as a trace of an error's origin.

## E

**EBCDIC.** (Extended Binary-Coded Decimal Interchange Code). A coded character set consisting of 8-bit coded characters.

**edit-directed transmission.** The type of stream-oriented transmission in which data appears as a continuous stream of characters and for which a format list is required to specify the editing desired for the associated data list.

**element.** A single item of data as opposed to a collection of data items such as an array; a scalar item.

**element expression.** An expression whose evaluation yields an element value.

**element variable.** A variable that represents an element; a scalar variable.

**elementary name.** See *base element*.

**enabled.** The state of a condition in which the condition can cause an interrupt and then invocation of the appropriate established ON-unit.

**entry constant.** (1) The label prefix of a PROCEDURE statement (an entry name). (2) The declaration of a name with the ENTRY attribute but without the VARIABLE attribute.

**entry data.** A data item that represents an entry point to a procedure.

**entry expression.** An expression whose evaluation yields an entry name.

**entry name.** (1) An identifier that is explicitly or contextually declared to have the ENTRY attribute (unless the VARIABLE attribute is given) or (2) An identifier that has the value of an entry variable with the ENTRY attribute implied.

**entry point.** A point in a procedure at which it can be invoked. *primary entry point* and *secondary entry point*.

**entry reference.** An entry constant, an entry variable reference, or a function reference that returns an entry value.

**entry variable.** A variable to which an entry value can be assigned. It must have both the ENTRY and VARIABLE attributes.

**entry value.** The entry point represented by an entry constant or variable; the value includes the environment of the activation that is associated with the entry constant.

**environment (of an activation).** Information associated with and used in the invoked block regarding data declared in containing blocks.

**environment (of a label constant).** Identity of the particular activation of a block to which a reference to a statement-label constant applies. This information is determined at the time a statement-label constant is passed as an argument or is assigned to a statement-label variable, and it is passed or assigned along with the constant.

**established action.** The action taken when a condition is raised. See also *implicit action* and *ON-statement action*.

**epilogue.** Those processes that occur automatically at the termination of a block or task.

**evaluation.** The reduction of an expression to a single value, an array of values, or a structured set of values.

**event.** An activity in a program whose status and completion can be determined from an associated event variable.

**event variable.** A variable with the EVENT attribute that can be associated with an event. Its value indicates whether the action has been completed and the status of the completion.

**explicit declaration.** The appearance of an identifier (a name) in a DECLARE statement, as a label prefix, or in a parameter list. Contrast with *implicit declaration*.

**exponent characters.** The following picture specification characters:

1. K and E, which are used in floating-point picture specifications to indicate the beginning of the exponent field.
2. F, the scaling factor character, specified with an integer constant that indicates the number of decimal positions the decimal point is to be moved from its assumed position to the right (if the constant is positive) or to the left (if the constant is negative).

**expression.** (1) A notation, within a program, that represents a value, an array of values, or a structured set of values; (2) A constant or a reference appearing alone, or a combination of constants and/or references with operators.

**extended alphabet.** The upper and lower case alphabetic characters A through Z, \$, @ and #, or those specified in the NAMES compiler option.

**extent.** (1) The range indicated by the bounds of an array dimension, by the length of a string, or by the size of an area (2) The size of the target area if this area were to be assigned to a target area.

**external name.** A name (with the EXTERNAL attribute) whose scope is not necessarily confined only to one block and its contained blocks.

**external procedure.** (1) A procedure that is not contained in any other procedure. (2) A level-2 procedure contained in a package that is also exported.

**extralingual character.** Characters (such as \$, @, and #) that are not classified as alphanumeric or

special. This group includes characters that are determined with the NAMES compiler option.

## F

**factoring.** The application of one or more attributes to a parenthesized list of names in a DECLARE statement, eliminating the repetition of identical attributes for multiple names

**field (in the data stream).** That portion of the data stream whose width, in number of characters, is defined by a single data or spacing format item.

**field (of a picture specification).** Any character-string picture specification or that portion (or all) of a numeric character picture specification that describes a fixed-point number.

**file.** A named representation, within a program, of a data set or data sets. A file is associated with the data set(s) for each opening.

**file constant.** A name declared with the FILE attribute but not the VARIABLE attribute.

**file description attributes.** Keywords that describe the individual characteristics of each file constant. See also *alternative attribute* and *additive attribute*.

**file expression.** An expression whose evaluation yields a value of the type file.

**file name.** A name declared for a file.

**file variable.** A variable to which file constants can be assigned. It has the attributes FILE and VARIABLE and cannot have any of the file description attributes.

**fixed-point constant.** See *arithmetic constant*.

**floating-point constant.** See *arithmetic constant*.

**flow of control.** Sequence of execution.

**format.** A specification used in edit-directed data transmission to describe the representation of a data item in the stream (data format item) or the specific positioning of a data item within the stream (control format item).

**format constant.** The label prefix on a FORMAT statement.

**format data.** A variable with the FORMAT attribute.

**format label.** The label prefix on a FORMAT statement.

**format list.** In stream-oriented transmission, a list specifying the format of the data item on the external medium. Contrast with *data list*.

**fully qualified name.** A name that includes all the names in the hierarchical sequence above the member to which the name refers, as well as the name of the member itself.

**function (procedure).** (1) A procedure that has a RETURNS option in the PROCEDURE statement. (2) A name declared with the RETURNS attribute. It is invoked by the appearance of one of its entry names in a function reference and it returns a scalar value to the point of reference. Contrast with *subroutine*.

**function reference.** An entry constant or an entry variable, either of which must represent a function, followed by a possibly empty argument list. Contrast with *subroutine call*.

## G

**generation (of a variable).** The allocation of a static variable, a particular allocation of a controlled or automatic variable, or the storage indicated by a particular locator qualification of a based variable or by a defined variable or parameter.

**generic descriptor.** A descriptor used in a GENERIC attribute.

**generic key.** A character string that identifies a class of keys. All keys that begin with the string are members of that class. For example, the recorded keys "ABCD," "ABCE," and "ABDF," are all members of the classes identified by the generic keys "A" and "AB," and the first two are also members of the class "ABC"; and the three recorded keys can be considered to be unique members of the classes "ABCD," "ABCE," "ABDF," respectively.

**generic name.** The name of a family of entry names. A reference to the generic name is replaced by the entry name whose parameter descriptors match the attributes of the arguments in the argument list at the point of invocation.

**group.** A collection of statements contained within larger program units. A group is either a do-group or a select-group and it can be used wherever a single statement can appear, except as an on-unit.

## H

**hex.** See *hexadecimal digit*.

**hexadecimal.** Pertaining to a numbering system with a base of sixteen; valid numbers use the digits 0 through 9 and the characters A through F, where A represents 10 and F represents 15.

**hexadecimal digit.** One of the digits 0 through 9 and A through F. A through F represent the decimal values 10 through 15, respectively.

## I

**identifier.** A string of characters, not contained in a comment or constant, and preceded and followed by a delimiter. The first character of the identifier must be one of the 26 alphabetic characters and extralingual characters, if any. The other characters, if any, can additionally include extended alphabetic, digit, or the break character.

**IEEE.** Institute of Electrical and Electronics Engineers.

**implicit.** The action taken in the absence of an explicit specification.

**implicit action.** The action taken when an enabled condition is raised and no ON-unit is currently established for the condition. Contrast with *ON-statement action*.

**implicit declaration.** A name not explicitly declared in a DECLARE statement or contextually declared.

**implicit opening.** The opening of a file as the result of an input or output statement other than the OPEN statement.

**infix operator.** An operator that appears between two operands.

**inherited dimensions.** For a structure, union, or element, those dimensions that are derived from the containing structures. If the name is an element that is not an array, the dimensions consist entirely of its inherited dimensions. If the name is an element that is an array, its dimensions consist of its inherited dimensions plus its explicitly declared dimensions. A structure with one or more inherited dimensions is called a nonconnected aggregate. Contrast with *connected aggregate*.

**input/output.** The transfer of data between auxiliary medium and main storage.

**insertion point character.** A picture specification character that is, on assignment of the associated data to a character string, inserted in the indicated position. When used in a P-format item for input, the insertion character is used for checking purposes.

**integer.** (1) An optionally signed sequence of digits or a sequence of bits without a decimal or binary point. (2) An optionally signed whole number, commonly described as FIXED BINARY (p,0) or FIXED DECIMAL (p,0).

**integral boundary.** A byte multiple address of any 8-bit unit on which data can be aligned. It usually is a half-word, full-word, or double-word (2-, 4-, or 8-byte multiple respectively) boundary.

**interleaved array.** An array that refers to nonconnected storage.

**interleaved subscripts.** Subscripts that exist in levels other than the lowest level of a subscripted qualified reference.

**internal block.** A block that is contained in another block.

**internal name.** A name that is known only within the block in which it is declared, and possibly within any contained blocks.

**internal procedure.** A procedure that is contained in another block. Contrast with *external procedure*.

**interrupt.** The redirection of the program's flow of control as the result of raising a condition or attention.

**invocation.** The activation of a procedure.

**invoke.** To activate a procedure.

**invoked procedure.** A procedure that has been activated.

**invoking block.** A block that activates a procedure.

**iteration factor.** (1) In an INITIAL attribute specification, an expression that specifies the number of consecutive elements of an array that are to be initialized with the given value. (2) In a format list, an expression that specifies the number of times a given format item or list of format items is to be used in succession.

**iterative do-group.** A do-group whose DO statement specifies a control variable and/or a WHILE or UNTIL option.

## K

**key.** Data that identifies a record within a direct-access data set. See *source key* and *recorded key*.

**keyword.** An identifier that has a specific meaning in PL/I when used in a defined context.

**keyword statement.** A simple statement that begins with a keyword, indicating the function of the statement.

**known (applied to a name).** Recognized with its declared meaning. A name is known throughout its scope.

## L

**label.** (1) A name prefixed to a statement. A name on a PROCEDURE statement is called an entry constant; a name on a FORMAT statement is called a format constant; a name on other kinds of statements is called a label constant. (2) A data item that has the LABEL attribute.

**label constant.** A name written as the label prefix of a statement (other than PROCEDURE, ENTRY, FORMAT, or PACKAGE) so that, during execution, program control can be transferred to that statement through a reference to its label prefix.

**label data.** A label constant or the value of a label variable.

**label prefix.** A label prefixed to a statement.

**label variable.** A variable declared with the LABEL attribute. Its value is a label constant in the program.

**leading zeroes.** Zeros that have no significance in an arithmetic value. All zeros to the left of the first nonzero in a number.

**level number.** A number that precedes a name in a DECLARE statement and specifies its relative position in the hierarchy of structure names.

**level-one variable.** (1) A major structure or union name. (2) Any unsubscripted variable not contained within a structure or union.

**lexically.** Relating to the left-to-right order of units.

**list-directed.** The type of stream-oriented transmission in which data in the stream appears as constants separated by blanks or commas and for which formatting is provided automatically.

**locator.** A control block that holds the address of a variable or its descriptor.

**locator/descriptor.** A locator followed by a descriptor. The locator holds the address of the variable, not the address of the descriptor.

**locator qualification.** In a reference to a based variable, either a locator variable or function reference connected by an arrow to the left of a based variable to specify the generation of the based variable to which the reference refers. It might be an implicit reference.

**locator value.** A value that identifies or can be used to identify the storage address.

**locator variable.** A variable whose value identifies the location in main storage of a variable or a buffer. It has the POINTER or OFFSET attribute.

**locked record.** A record in an EXCLUSIVE DIRECT UPDATE file that has been made available to one task only and cannot be accessed by other tasks until the task using it relinquishes it.

**logical level (of a structure or union member).** The depth indicated by a level number when all level numbers are in direct sequence (when the increment between successive level numbers is one).

**logical operators.** The bit-string operators not and exclusive-or ( $\sim$ ), and ( $\&$ ), and or ( $\mid$ ).

**loop.** A sequence of instructions that is executed iteratively.

**lower bound.** The lower limit of an array dimension.

## M

**main procedure.** An external procedure whose PROCEDURE statement has the OPTIONS (MAIN) attribute. This procedure is invoked automatically as the first step in the execution of a program.

**major structure.** A structure whose name is declared with level number 1.

**member.** A structure, union, or element name, possibly dimensioned, in a structure or union.

**minor structure.** A structure that is contained within another structure or union. The name of a minor structure is declared with a level number greater than one and greater than its parent structure or union.

**mode (of arithmetic data).** An attribute of arithmetic data. It is either *real* or *complex*.

**multiple declaration.** (1) Two or more declarations of the same identifier internal to the same block without different qualifications. (2) Two or more external declarations of the same identifier.

**multiprocessing.** The use of a computing system with two or more processing units to execute two or more programs simultaneously.

**multiprogramming.** The use of a computing system to execute more than one program concurrently, using a single processing unit.

**multitasking.** A facility that allows a program to execute more than one PL/I procedure simultaneously.

## N

**name.** Any identifier that the user gives to a variable or to a constant. An identifier appearing in a context where it is not a keyword. Sometimes called a user-defined name.

**nesting.** The occurrence of:

- A block within another block
- A group within another group
- An IF statement in a THEN clause or in an ELSE clause
- A function reference as an argument of a function reference
- A remote format item in the format list of a FORMAT statement
- A parameter descriptor list in another parameter descriptor list
- An attribute specification within a parenthesized name list for which one or more attributes are being factored.

**nonconnected storage.** Storage occupied by nonconnected data items. For example, interleaved arrays and structures with inherited dimensions are in nonconnected storage.

**null locator value.** A special locator value that cannot identify any location in internal storage. It gives a positive indication that a locator variable does not currently identify any generation of data.

**null statement.** A statement that contains only the semicolon symbol (;). It indicates that no action is to be taken.

**null string.** A character, graphic, or bit string with a length of zero.

**numeric-character data.** See *decimal picture data*.

**numeric picture data.** Picture data that has an arithmetic value as well as a character value. This type of picture data cannot contain the characters "A" or "X."

## O

**object.** A collection of data referred to by a single name.

**offset variable.** A locator variable with the OFFSET attribute, whose value identifies a location in storage relative to the beginning of an area.

**ON-condition.** An occurrence, within a PL/I program, that could cause a program interrupt. It can be the detection of an unexpected error or of an occurrence that is expected, but at an unpredictable time.

**ON-statement action.** The action explicitly established for a condition that is executed when the condition is raised. When the ON-statement is encountered in the flow of control for the program, it executes, establishing the action for the condition. The action executes when the condition is raised if the ON-unit is still established or a RESIGNAL statement re-establishes it. Contrast with *implicit action*.

**ON-unit.** The specified action to be executed when the appropriate condition is raised.

**opening (of a file).** The association of a file with a data set.

**operand.** The value of an identifier, constant, or an expression to which an operator is applied, possibly in conjunction with another operand.

**operational expression.** An expression that consists of one or more operators.

**operator.** A symbol specifying an operation to be performed.

**option.** A specification in a statement that can be used to influence the execution or interpretation of the statement.

## P

**package constant.** The label prefix on a PACKAGE statement.

**packed decimal.** The internal representation of a fixed-point decimal data item.

**padding.** (1) One or more characters, graphics, or bits concatenated to the right of a string to extend the string to a required length. (2) One or more bytes or bits inserted in a structure or union so that the following element within the structure or union is aligned on the appropriate integral boundary.

**parameter.** A name in the parameter list following the PROCEDURE statement, specifying an argument that will be passed when the procedure is invoked.

**parameter descriptor.** The set of attributes specified for a parameter in an ENTRY attribute specification.

**parameter descriptor list.** The list of all parameter descriptors in an ENTRY attribute specification.

**parameter list.** A parenthesized list of one or more parameters, separated by commas and following either the keyword PROCEDURE in a procedure statement or the keyword ENTRY in an ENTRY statement. The list corresponds to a list of arguments passed at invocation.

**partially qualified name.** A qualified name that is incomplete. It includes one or more, but not all, of the names in the hierarchical sequence above the structure or union member to which the name refers, as well as the name of the member itself.

**picture data.** Numeric data, character data, or a mix of both types, represented in character form.

**picture specification.** A data item that is described using the picture characters in a declaration with the PICTURE attribute or in a P-format item.

**picture specification character.** Any of the characters that can be used in a picture specification.

**PL/I character set.** A set of characters that has been defined to represent program elements in PL/I.

**point of invocation.** The point in the invoking block at which the reference to the invoked procedure appears.

**pointer.** A type of variable that identifies a location in storage.

**pointer value.** A value that identifies the pointer type.

**pointer variable.** A locator variable with the POINTER attribute that contains a pointer value.

**precision.** The number of digits or bits contained in a fixed-point data item, or the minimum number of significant digits (excluding the exponent) maintained for a floating-point data item.

**prefix.** A label or a parenthesized list of one or more condition names included at the beginning of a statement.

**prefix operator.** An operator that precedes an operand and applies only to that operand. The prefix operators are plus (+), minus (-), and not (~).

**preprocessor.** A program that examines the source program before the compilation takes place.

**preprocessor statement.** A special statement appearing in the source program that specifies the actions to be performed by the preprocessor. It is executed as it is encountered by the preprocessor.

**primary entry point.** The entry point identified by any of the names in the label list of the PROCEDURE statement.

**priority.** A value associated with a task, that specifies the precedence of the task relative to other tasks.

**problem data.** Coded arithmetic, bit, character, graphic, and picture data.

**problem-state program.** A program that operates in the problem state of the operating system. It does not contain input/output instructions or other privileged instructions.

**procedure.** A collection of statements, delimited by PROCEDURE and END statements. A procedure is a program or a part of a program, delimits the scope of names, and is activated by a reference to the procedure or one of its entry names. See also *external procedure* and *internal procedure*.

**procedure reference.** An entry constant or variable. It can be followed by an argument list. It can appear in a CALL statement or the CALL option, or as a function reference.

**program.** A set of one or more external procedures or packages. One of the external procedures must have the OPTIONS(MAIN) specification in its procedure statement.

**program control data.** Area, locator, label, format, entry, and file data that is used to control the processing of a PL/I program.

**prologue.** The processes that occur automatically on block activation.

**pseudovisible.** Any of the built-in function names that can be used to specify a target variable. It is usually on the left-hand side of an assignment statement.

## Q

**qualified name.** A hierarchical sequence of names of structure or union members, connected by periods, used to identify a name within a structure. Any of the names can be subscripted.

## R

**range (of a default specification).** A set of identifiers and/or parameter descriptors to which the attributes in a DEFAULT statement apply.

**record.** (1) The logical unit of transmission in a record-oriented input or output operation. (2) A collection of one or more related data items. The items usually have different data attributes and usually are described by a structure or union declaration.

**recorded key.** A character string identifying a record in a direct-access data set where the character string itself is also recorded as part of the data.

**record-oriented data transmission.** The transmission of data in the form of separate records. Contrast with *stream data transmission*.

**recursive procedure.** A procedure that can be called from within itself or from within another active procedure.

**reentrant procedure.** A procedure that can be activated by multiple tasks, threads, or processes simultaneously without causing any interference between these tasks, threads, and processes.

**REFER expression.** The expression preceding the keyword REFER, which is used as the bound, length, or size when the based variable containing a REFER option is allocated, either by an ALLOCATE or LOCATE statement.

**REFER object.** The variable in a REFER option that holds or will hold the current bound, length, or size for the member. The REFER object must be a member of the same structure or union. It must not be locator-qualified or subscripted, and it must precede the member with the REFER option.

**reference.** The appearance of a name, except in a context that causes explicit declaration.

**relative virtual origin (RVO).** The actual origin of an array minus the virtual origin of an array.

**remote format item.** The letter R followed by the label (enclosed in parentheses) of a FORMAT statement. The format statement is used by edit-directed data transmission statements to control the format of data being transmitted.

**repetition factor.** A parenthesized unsigned integer constant that specifies:

1. The number of times the string constant that follows is to be repeated.

2. The number of times the picture character that follows is to be repeated.

**repetitive specification.** An element of a data list that specifies controlled iteration to transmit one or more data items, generally used in conjunction with arrays.

**restricted expression.** An expression that can be evaluated by the compiler during compilation, resulting in a constant. Operands of such an expression are constants, named constants, and restricted expressions.

**returned value.** The value returned by a function procedure.

**RETURNS descriptor.** A descriptor used in a RETURNS attribute, and in the RETURNS option of the PROCEDURE and ENTRY statements.

## S

**scalar variable.** A variable that is not a structure, union, or array.

**scale.** A system of mathematical notation whose representation of an arithmetic value is either fixed-point or floating-point.

**scale factor.** A specification of the number of fractional digits in a fixed-point number.

**scaling factor.** See *scale factor*.

**scope (of a condition prefix).** The portion of a program throughout which a particular condition prefix applies.

**scope (of a declaration or name).** The portion of a program throughout which a particular name is known.

**secondary entry point.** An entry point identified by any of the names in the label list of an entry statement.

**select-group.** A sequence of statements delimited by SELECT and END statements.

**selection clause.** A WHEN or OTHERWISE clause of a select-group.

**self-defining data.** An aggregate that contains data items whose bounds, lengths, and sizes are determined at program execution time and are stored in a member of the aggregate.

**separator.** See *delimiter*.

**shift.** Change of data in storage to the left or to the right of original position.

**shift-in.** Symbol used to signal the compiler at the end of a double-byte string.

**shift-out.** Symbol used to signal the compiler at the beginning of a double-byte string.

**sign and currency symbol characters.** The picture specification characters. S, +, -, and \$ (or other national currency symbols enclosed in < and >).

**simple parameter.** A parameter for which no storage class attribute is specified. It can represent an argument of any storage class, but only the current generation of a controlled argument.

**simple statement.** A statement other than IF, ON, WHEN, and OTHERWISE.

**source.** Data item to be converted for problem data.

**source key.** A key referred to in a record-oriented transmission statement that identifies a particular record within a direct-access data set.

**source program.** A program that serves as input to the source program processors and the compiler.

**source variable.** A variable whose value participates in some other operation, but is not modified by the operation. Contrast with *target variable*.

**standard default.** The alternative attribute or option assumed when none has been specified and there is no applicable DEFAULT statement.

**standard file.** A file assumed by PL/I in the absence of a FILE or STRING option in a GET or PUT statement. SYSIN is the standard input file and SYSPRINT is the standard output file.

**standard system action.** Action specified by the language to be taken for an enabled condition in the absence of an on-unit for that condition.

**statement.** A PL/I statement, composed of keywords, delimiters, identifiers, operators, and constants, and terminated by a semicolon (;). Optionally, it can have a condition prefix list and a list of labels. See also *keyword statement*, *assignment statement*, and *null statement*.

**statement body.** A statement body can be either a simple or a compound statement.

**statement label.** See *label constant*.

**static storage allocation.** The allocation of storage for static variables.

**static variable.** A variable that is allocated before execution of the program begins and that remains allocated for the duration of execution.

**stream-oriented data transmission.** The transmission of data in which the data is treated as though it were a continuous stream of individual data values in character form. Contrast with *record-oriented data transmission*.

**string.** A contiguous sequence of characters, graphics, or bits that is treated as a single data item.

**string variable.** A variable declared with the BIT, CHARACTER, or GRAPHIC attribute, whose values can be either bit, character, or graphic strings.

**structure.** A collection of data items that need not have identical attributes. Contrast with *array*.

**structure expression.** An expression whose evaluation yields a structure set of values.

**structure of arrays.** A structure that has the dimension attribute.

**structure member.** See *member*.

**structuring.** The hierarchy of a structure, in terms of the number of members, the order in which they appear, their attributes, and their logical level.

**subroutine.** A procedure that has no RETURNS option in the PROCEDURE statement. Contrast with *function*.

**subroutine call.** An entry reference that must represent a subroutine, followed by an optional argument list that appears in a CALL statement. Contrast with *function reference*.

**subscript.** An element expression that specifies a position within a dimension of an array. If the subscript is an asterisk, it specifies all of the elements of the dimension.

**subscript list.** A parenthesized list of one or more subscripts, one for each dimension of the array, which together uniquely identify either a single element or cross section of the array.

**subtask.** A task that is attached by the given task or any of the tasks in a direct line from the given task to the last attached task.

**synchronous.** A single flow of control for serial execution of a program.

## T

**target.** Attributes to which a data item (source) is converted.

**target reference.** A reference that designates a receiving variable (or a portion of a receiving variable).

**target variable.** A variable to which a value is assigned.

**task.** The execution of one or more procedures by a single flow of control .

**task name.** An identifier used to refer to a task variable.

**task variable.** A variable with the TASK attribute whose value gives the relative priority of a task.

**termination (of a block).** Cessation of execution of a block, and the return of control to the activating block by means of a RETURN or END statement, or the transfer of control to the activating block or to some other active block by means of a GO TO statement.

**termination (of a task).** Cessation of the flow of control for a task.

**truncation.** The removal of one or more digits, characters, graphics, or bits from one end of an item of data when a string length or precision of a target variable has been exceeded.

**type.** The set of data attributes and storage attributes that apply to a generation, a value, or an item of data.

## U

**undefined.** Indicates something that a user must not do. Use of an undefined feature is likely to produce different results on different implementations of a PL/I product. In that case, the application program is considered to be in error.

**union.** A collection of data elements that overlay each other, occupying the same storage. The members can be structures, unions, elementary variables, or arrays. They need not have identical attributes.

**union of arrays.** A union that has the DIMENSION attribute.

**upper bound.** The upper limit of an array dimension.

## V

**value reference.** A reference used to obtain the value of an item of data.

**variable.** A named entity used to refer to data and to which values can be assigned. Its attributes remain constant, but it can refer to different values at different times.

**variable reference.** A reference that designates all or part of a variable.

**virtual origin (VO).** The location where the element of the array whose subscripts are all zero are held. If such an element does not appear in the array, the virtual origin is where it would be held.

## Z

**zero-suppression characters.** The picture specification characters Z and \*, which are used to suppress zeros in the corresponding digit positions and replace them with blanks or asterisks respectively.

---

# Index

## Special Characters

- \*PROCESS, specifying options in 28
- %INCLUDE statement, incorporating source code into program 62
- %PATHCODE value associated with hook, querying 271
- %PROCESS, specifying options in 28

## A

- abbreviation
  - compile-time option 5
- abend
  - ABEND80A 69
  - during in-line input/output 151
- access
  - ESDS 231
  - indexed data set 179
    - direct access 181
    - sequential access 180
  - regional data set 208
  - REGIONAL(1) data set 192, 194
    - direct access 193
    - sequential access 193
  - REGIONAL(2) data set 198
    - direct access 199
    - directly 200
    - sequential access 199
    - sequentially 201
  - REGIONAL(3) data set 204
    - direct access 205
    - directly 206
    - sequential access 204
    - sequentially 207
  - relative-record data set 252
- access method service
  - AMSERV command 83
- accessing data sets by a single statement 340
- ACCT EXEC statement parameter 54
- activating hooks
  - in compiled programs 265—266
  - using IBMBHKS 265—266
- actual
  - task hierarchy 427
- ADDBUFF option
  - ENVIRONMENT option 111
  - indexed data set 169
- adding records for indexed data set 168
- ADDR
  - ESD heading 42
- address parameter 32
- addresses
  - area length 381
  - area starting 381
  - argument list 373
  - array descriptor 381
  - start of the array or structure 381
  - strings 383
  - structure descriptor 381
- aggregate
  - AGGREGATE option 8
  - length table 38
- aggregate locators
  - array descriptor addresses 381
  - array starting addresses 381
  - contents of 381
  - format of 381
  - structure descriptor addresses 381
  - structure starting addresses 381
- aggregates, locator 381
- aliased variables
  - inhibiting optimization 318
  - optimization, inhibiting 318
- ALIGNED attribute 325
- ALL option
  - hooks location suboption 26
- ALLOCATE statement 39
- allocating, registers, effect of REORDER option 315, 317
- allocation
  - base register for branch instructions 311
  - data sets for compilation 57
  - of buffers 339
- alternate index path
  - KSDS 242, 246
  - nonunique key 240
    - ESDS 241
    - VSAM 242
  - unique key
    - VSAM 240
  - VSAM 223
    - ESDS 242, 243
    - KSDS 243
    - process 218
- alternative MAIN, invocation of 398
- American National Standard (ANS) control characters 16
- AMP parameter 211
- AMSERV command 83
- analyzing CPU-time usage (example)
  - discussion of programs 288
  - output from 288—289

- analyzing CPU-time usage (example) *(continued)*
  - setup for 288
  - source code for 290—304
- apparent task hierarchy 427
- applications
  - tuning
    - for virtual storage system 307, 308
    - source code 305
- area descriptors
  - concatenation with array descriptor 382
  - in structure descriptors 382
- area locator/descriptors
  - area length addresses 381
  - area starting addresses 381
  - area variables in 382
  - contents of 381
  - format of 382
- AREA ON-unit, avoiding indefinite loop 345
- area variables
  - in area locator/descriptors 382
- areas
  - length addresses 381
  - locator/descriptor 381
  - overflow 167
  - prime data 167
  - starting addresses 381
- argument lists
  - addresses of 373
  - storage for 373
- arguments
  - data descriptors 373
  - passing 373
  - sort program 355
- array descriptors
  - bounds components 383
  - concatenation of string or area descriptor with 382
  - contents of 382
  - format of
    - with CMPAT(V1) option 382
    - with CMPAT(V2) option 382
  - multiplier components 383
  - relative virtual origin component 383
- array elements
  - inhibiting optimization 319
  - multiplication operations
    - loop optimization 310, 312
    - optimization by repeated addition 310, 312
    - optimization of subscript expressions 311
- array of areas
  - concatenation of area descriptor with array descriptor 382
- array of strings
  - concatenation of string descriptor with array descriptor 382
- array subscripts
  - FIXED BINARY data type 324
- array subscripts *(continued)*
  - optimization of constants in expressions 311
- arrays
  - array arithmetic 327
  - array descriptors, contents of 382
  - assignments, optimization of 313
  - base element offsets 384
  - common control data, elimination of 313
  - dimension multiplier 382
  - element assignments 313
  - elimination of common control data 313
  - initialization, optimization of 313
  - names in data list 342
  - of structures
    - aggregate locators for 385
    - structure descriptors for 385
  - optimization of
    - element assignments 313
    - initialization 313
    - relative virtual origin (RVO) 382
    - restriction on INITIAL attribute 333
    - storage location 381
- arrays, first element of, unaligned bit string restriction 339
- ASCII 154
  - ENVIRONMENT option 111
  - option of ENVIRONMENT
    - comparison with DCB subparameter 113
    - for stream I/O 130
  - records 106
- assembler routines
  - argument list addresses
    - table of 373
    - with OPTIONS(ASSEMBLER) attribute 374
    - without OPTIONS(ASSEMBLER) attribute 374
  - calls from PL/I to 374
  - descriptors 373
  - invoking 374
  - invoking the compiler 32
    - ddname list 33
    - option list 33
    - page number 33
  - locators 373
  - OPTIONS(ASSEMBLER) attribute 374
  - passing arguments and receiving parameters 374
  - passing pointers 375
  - recursive 332
  - simulating a function reference 374
- assignments
  - array and structure, optimized 313
- associating
  - data sets with files 99
  - data sets with one file 102
  - files with one data set 101
- ATTACH macro instruction 32

- attention handler, return code 412
- attention interrupt
  - effect under interactive system 13
- ATTENTION ON-units 437
- attention processing
  - ATTENTION ON-units 437
  - debugging tool 437
  - main description 436
  - passing control to PLITEST 437
- attention router
  - preinitialized program 413
    - example of 413
  - return/reason codes 412
- attention router service routine
  - preinitialized program 412
- attribute table 37
- attribute table description 79
- ATTRIBUTES option 8
- automatic
  - padding 88, 91
  - restart
    - after system failure 440
    - checkpoint/restart facility 438
    - within a program 441
  - variable location 15
- automatic prompting
  - overriding 87, 90
  - using 86, 90
- AUTOMATIC variables, allocation of 333
- automatic variables, storage allocation for 331
- auxiliary storage for sort 355

## B

- background region 63
- background region, running jobs in 63
- backward reading with ESDES
- BACKWARDS attribute 158
- base register allocation for branch instructions 311
- based and controlled variables, assigning storage
  - classes for a virtual storage system 307
- based variables
  - inhibiting optimization 318
  - optimization, inhibiting 318
- batched compilation
  - examples of 71
  - JCL 70
  - MVS 55, 64
  - problems with OBJECT, MDECK, and DECK 69
  - return code 70
  - storage abend 69
  - VM 78
- BEGIN statement 38
- begin-blocks, effect of ORDER option on 316
- BINARYVALUE 14

- bit string arrays
  - bit offset of 383
- bit strings
  - eight-bit multiples 324
  - used as logical switches 324
- BKWD option 111, 220
- BLKSIZE 108, 115
  - consecutive data sets 156, 157
  - DCB subparameter
    - indexed data set 174
  - ENVIRONMENT 111
    - for record I/O 115
    - option of ENVIRONMENT
      - comparison with DCB subparameter 113
      - for stream I/O 130
- BLOCK hooks location suboption 26
- block information control block
  - layout of 270
  - specifying pointer to 268
- block size 116
  - consecutive data sets 156
  - indexed data sets 173
  - object module 65
  - PRINT files 139
  - record length 116
  - regional data sets 209
  - specifying 103
- blocks
  - nested, declaring arithmetic variables 316
  - querying block number 271
- blocks and records 103
- branch instructions, base register allocation 311
- BUFFERS 130
- BUFFERS option 111, 117, 258
  - comparison with DCB subparameter 113
  - for stream I/O 130
- buffers, allocation of 339
- BUFND option 220
- BUFND option of ENVIRONMENT 111
- BUFNI option 220
- BUFNI option of ENVIRONMENT 111
- BUFNO DCB subparameter
  - consecutive data set 157
  - indexed data set 174
- BUFNO subparameter 108
- BUFOFF 155
- BUFOFF option of ENVIRONMENT 111
  - comparison with DCB subparameter 113
  - for stream I/O 130
- BUFSP option 221
- BUFSP option of ENVIRONMENT 111
- built-in functions
  - in-line code for 314
- BY expression
  - bounds of, computation 334
  - computation of bounds 334

BYVALUE option 378

## C

CALL extended parameter list request 389

CALL macro instruction 32

CALL statement

  multitasking 423

    EVENT option 423

    PRIORITY option 424

    TASK option 423

callable services

  IBMBHKS 265

  IBMBSIR 267

  IBMHSIR 271

calling

  sort program 355

    establishing data sets 358

capacity record

  REGIONAL(1) 191

  REGIONAL(2) 197

  REGIONAL(3) 202

case for record files 146

case for stream files 145

cataloged procedure

  multitasking 53

cataloged procedures

  compile and link-edit 48

  compile only 47

  compile, input data for 50

  compile, link-edit, and run 50

  compile, load and run 51

  description of 46

  invoking 52

  listing 52

  modifying

    DD statement 55

    EXEC statement 54

  multiple invocations 52

  under MVS

    IBM-supplied 46

    to invoke 52

    to modify 54

CEEIBINT

  analyzing CPU-time usage example

    source code for 290—296

    uses of 288

  code coverage example

    source code for 274—279

    uses of 272

  function trace example

    source code for 285—286

    uses of 284

  use with IBMBHIR 265, 272

  use with IBMBHKS 265, 272

  use with IBMBSIR 265, 272

CEEIBINT (*continued*)

  using hook exit in 265, 272

CEEFMAIN

  control section 43

CEESTART

  ESD entry 42

character string attribute table 37

checkpoint data set for sort 360

checkpoint data, defining, PLICKPT built-in  
  suboption 440

checkpoint record, PLICKPT built-in subroutine 439

checkpoint/restart

  CALL PLIREST statement 441

  checkpoint data set 440

  deferred restart 441

  description of 438

  modify activity 441

  PLICANC statement 441

  PLICKPT built-in subroutine 438

  request checkpoint record 439

  request restart 440

  RESTART parameter 441

  return codes 439

CHKPT sort option 353

choosing type of sort 349

CICS, compiling transactions in PL/I 72

CKPT sort option 353

CLOSE statement 424

CLOSE statements, specifying more than one file 340

CMPAT

  compile-time option 8

CMPAT(V1) option

  format of array descriptor with 382

CMPAT(V2) option

  declaring control variables 335

  format of array descriptor with 382

COBOL

  COBOL option 120

  data interchange 120

  map structure 39

  mapped structure, compiler listing 39

  option of the ENVIRONMENT attribute 111

    VSAM data sets 219

  structures in aggregate length table 39

code

  coverage, checking via the hook exit 272

  for program branches 311

  in-line

    for built-in functions 314

    for conversions 314

    for record I/O 314

    for string manipulation 314

CODE subparameter 108

  consecutive data set 157

common area ESD entry 41

- common constants
  - elimination of 311
  - optimization of 311
- common control data
  - elimination of 313
- common expression
  - definition of 309
  - elimination of 309
  - example of 309
- common expression elimination 309, 318, 319, 320
- compatibility
  - arrays, AREAS, aggregates
    - using CMPAT 8
  - fullword subscript 8
  - object 8
  - version 1 8
  - VSAM interface 225
- compilation, and size of programs 331
- compile and link-edit, input data for 48
- COMPILE option 10
- compile-time options
  - abbreviations 5
  - AGGREGATE 8
  - ATTRIBUTES 8
  - CMPAT 8
  - COMPILE 10
  - CONTROL 10
  - DECK 10
  - default 5
  - description of 5
  - ESD 11
  - FLAG 11
  - GONUMBER 11
  - GONUMBER, storage requirements for 306
  - GOSTMT 12
  - GOSTMT, storage requirements for 306
  - GRAPHIC 12
  - IMPRECISE 12
  - INCLUDE 13
  - INSOURCE 13
  - INTERRUPT 13
  - LANGLVL 14
  - LINECOUNT 14
  - LIST 14
  - LMESSAGE 15
  - MACRO 15
  - MAP 15
  - MARGINI 16
  - MARGINS 16
  - MDECK 17
  - NAME 17
  - NEST 18
  - NUMBER 18
  - OBJECT 19
  - OFFSET 19
  - OPTIMIZE 20

- compile-time options (*continued*)
  - OPTIONS 20
  - SEQUENCE 21
  - SIZE 22
  - SMESSAGE 23
  - SOURCE 23
  - STMT 23
  - STORAGE 23
  - SYNTAX 24
  - SYSTEM 24
  - TERMINAL 25
  - TEST 26
  - XREF 27
- compiler
  - % statements
    - description of 32
    - using 32
  - check out program option 78
  - correcting errors 71, 78
  - data sets 58
  - DBCS identifier 12
  - ddname list 33
  - descriptions of options 5
  - error correction 78
  - EXEC statement 64
  - from an assembler routine 32
  - general description of 55
  - graphic string constant 12
  - input record format 28
  - input record limit 16
  - invoking from an assembler routine 32
  - JCL statements, using 64
  - limit storage size 22
  - listing
    - aggregate length table 38
    - attribute table 37
    - block level 36
    - COBOL mapped structure 39
    - cross-reference table 37
    - DO-level 36
    - error messages 11
    - external symbol dictionary (ESD) 41
    - heading information 34
    - include source program 13
    - input to compiler 34
    - input to preprocessor 34
    - main storage requirement 23
    - messages 44
    - number of lines per page 14
    - object module 44
    - printing options 62, 67
    - return codes 45
    - SOURCE program 23, 34
    - statement offset addresses 40
    - static internal storage map 43
    - storage requirements 39
    - SYSPRINT 67

- compiler (*continued*)
  - listing (*continued*)
    - TSO 55, 62
    - using 33
    - variable offset map 43
  - minimum region size
  - mixed string constant 12
  - option list 33
  - passing address parameters to 32
  - PLIOPT command 73
  - preprocessor 29
  - PROCESS statement 28
  - reduce storage requirement 20
  - reinstate options deleted from installation 10
  - severity of error condition 10
  - suppressing in the case of error 79
  - temporary workfile (SYSUT1) 66
  - under MVS 56
  - under MVS batch 64
  - under TSO 56
  - under VM 73
  - VM to run under MVS 77
- compiler listings
  - directing to data set 62
  - source listing at terminal 62
- compiler page numbering 33
- compiler, MVS/XA 71
- compiling
  - CICS transactions in PL/I 72
  - under TSO
    - data sets 57
    - PLI command 56, 59
- complex expressions, rules for precision 325
- concatenating
  - data sets 102
  - external references 100
- COND EXEC statement parameter 54
- condition handling
  - common expression elimination 320
  - performance improvement 344
  - teleprocessing data sets 260
- conditional
  - compilation 10
  - subparameter 108
- conditions
  - disabled, required processing 344
  - disabling debugging 344
  - teleprocessing 260
- consecutive data sets
  - controlling input from the terminal
    - capital and lowercase letters 145
    - conditions format 143
    - COPY option of GET statement 145
    - end-of-file 145
    - format of data 143
    - stream and record files 144
    - using files conversationally 143
  - consecutive data sets (*continued*)
    - controlling output to the terminal
      - capital and lowercase letters 146
      - conditions 145
      - example of 146
      - format of PRINT files 145
      - output from the PUT EDIT command 146
      - stream and record files 146
    - defining and using 129
    - input from the terminal 143
    - output to the terminal 145
    - record-oriented data transmission
      - accessing and updating a data set 157
      - creating a data set 156
      - defining files 150
      - specifying ENVIRONMENT options 150
      - statements and options allowed 149
    - record-oriented I/O 149
    - stream-oriented data transmission 129
      - accessing a data set 136
      - creating a data set 132
      - defining files 130
      - specifying ENVIRONMENT options 130
      - using PRINT files 138
      - using SYSIN and SYSPRINT files 142
- CONSECUTIVE data sets, generating in-line code 340
- CONSECUTIVE file
  - adapting for VSAM 226
  - compatibility with VSAM 224
- CONSECUTIVE option 130, 151
- constant exponents, replacement of 311
- constant expressions
  - optimization of 311
  - replacement of 311
- constant multipliers, replacement of 311
- constants
  - common
    - elimination of 311
    - optimization of 311
  - exponents, optimization of 311
  - expressions
    - optimization of 311
    - transferring outside of loops 312
  - multipliers, optimization of 311
  - optimization of common constants 311
  - optimization of constant exponents 311
  - optimization of constant expressions 311
  - optimization of constant multipliers 311
  - statements, transferring outside of loops 312
- continuation line for compile-time options 68
- control
  - areas 212
  - characters 138
  - CONTROL option
    - compile-time 10
    - EXEC statement 68

- control (*continued*)
  - intervals 212
- control blocks
  - descriptors 380
  - locators 380
- control characters, restriction on use 325
- control data, common, elimination of 313
- controlled variables
  - area sizes of 333
  - array bounds of 333
  - string lengths of 333
- conversational
  - using files 143
- conversational programs, creating 89
- conversions
  - avoiding, use of additional variables for 326
- conversions, in-line code for 314
- coordination for multitasking 424
- COPY option
  - GET statement 145
- correcting error conditions, effect of REORDER option on 317
- corresponding data sets 106
- creating
  - REGIONAL(1) data set 191, 192
  - REGIONAL(2) data set 197, 198
  - REGIONAL(3) data set 202, 203
- cross-reference table 36, 37
  - description of 79
- CTLASA and CTL360 152
  - ENVIRONMENT option 111
    - comparison with DCB subparameter 113
    - for consecutive data sets 152
    - SCALARVARYING 121
- cylinder
  - index 166
  - overflow area 167, 177
- CYLOFL subparameter 108
  - indexed data set 174
  - overflow area 177

## D

- D option of ENVIRONMENT
  - for stream I/O 130
- D-format and DB-format records 155
- D-format records 155
- data
  - conversion for performance improvement 329
  - declarations, external 332
  - descriptors
    - data element descriptors 380
    - descriptors and locators 380
    - passing arguments and returned values 373
    - terminology 380
  - elements for performance improvement 323

- data (*continued*)
  - external declarations 332
  - relation to locators and descriptors 380
  - share between tasks 425
  - sort program 360
    - description of 348
    - PLISRT(x) command 365
    - sorting 348
- data conversions
  - arithmetic, allowable picture characters for in-line operations 329
  - in-line operations for
    - allowable picture characters for arithmetic 329
    - table of 329
  - performed in-line 329
    - table of 329
- data lists, matched with format lists in edit-directed transmission 315
- data sets
  - adapt existing program
    - CONSECUTIVE file 226
    - INDEXED file 226
    - REGIONAL(1) file 226
  - allocating for compilation 57
  - alternate index path with a file 211
  - ASCII 106
  - associating data sets with files 99
  - associating one data set with several files 101
  - associating PL/I files with
    - closing a file 110
    - opening a file 109
    - specifying characteristics in the ENVIRONMENT attribute 110
  - associating several data sets with one file 102
  - blocks 103
  - blocks and records 103
  - checkpoint 440
    - defining 440
  - closing 110
  - concatenating several 102
  - conditional subparameter characteristics 108
  - consecutive 129
  - consecutive stream-oriented data 129
  - containing secondary input, allocating 63
  - data set control block (DSCB) 107
- data sets
  - alternate index paths 223
  - blocking 212
  - choosing a type 216
  - defining files for 218
  - dummy data set 217
  - file attribute 218
  - keys for 215
  - performance options 223
  - running a program with 211
  - specifying ENVIRONMENT options 219
  - types of 214

data sets (*continued*)

- ddnames 64
- defining data sets 99
- defining for dumps
  - DD statement 387
  - logical record length 387
- defining relative-record 249
- direct 106, 107
- dissociating from a file 110
- dissociating from PL/I file 102
- establishing characteristics 102
- files defined for non-VSAM data set
  - adapting existing programs 225
  - compatibility interface 225
  - CONSECUTIVE files 224
  - INDEXED files 224
  - several files in one VSAM data set 226
  - shared data sets 227
- independent overflow area 177
- indexed 172
  - defining and using 163
  - master index 178
  - name 175
  - overflow area 177
  - record format 175
  - sequential 106
- indexed data set
  - load statement and options 232
- information interchange codes 103
- input in cataloged procedures 46
- label modification 109
- labels 107
  - in library data sets 123
- libraries
  - extracting information 128
  - SPACE parameter 124
  - types of 123
  - updating 125
  - use 123
- mass sequential insert 238
- master index 178
- organization
  - conditional subparameters 108
  - data definition (DD) statement 107
  - labels 107
  - types of 106
- OS-simulated 82
- partitioned 123
- PLI command to name 59
- record format defaults 113
- record formats
  - fixed-length 104
  - undefined-length 106
  - variable-length 104
- records 103
- regional 185

data sets (*continued*)

- REGIONAL, key handling optimization 314
- REGIONAL(1) 190
  - accessing and updating 192
  - creating 191
- REGIONAL(2) 195
  - accessing and updating 198
  - creating 197
  - keys 195
- REGIONAL(3) 202
  - accessing and updating 204
  - creating 202
- relative-record data set 249
- sequential 106
- sort program 358
  - checkpoint data set 360
  - input data set 359
  - output data set 359
  - sort work data set 359
  - SORTWK 355
- source statement library 67
- SPACE parameter 64
- stream files 129
- teleprocessing
  - define file 257
  - ENVIRONMENT options 257
- temporary 66
- to establish characteristics 102
- TRANSIENT 107, 256
- types of
  - comparison 122
  - organization 106
  - used by PL/I record I/O 121
- unlabeled 107
- using 99
- VSAM
  - data set type 216
  - defining 227
  - defining files 218
  - keys 215
  - running a program 211
  - several files in one data set 226
  - VSAM option 223
- DATE built-in function 338
- DB option of ENVIRONMENT
  - for stream I/O 130
- DB-format records 155
- DBCS identifier compilation 12
- DBCSOS ordering product 37
  - under MVS 23
  - under VM 22
- DCB subparameter 110, 113
  - consecutive data sets 157
  - equivalent ENVIRONMENT options 113
  - indexed data set 174
  - main discussion of 108

DCB subparameter (*continued*)  
 overriding in cataloged procedure 55  
 regional data set 210

DD statement 107, 133, 136  
 %INCLUDE 31  
 add to cataloged procedure 55  
 AMP parameter 211  
 cataloged procedure, modifying 55  
 checkpoint/restart 439  
 create a library 124  
 indexed data set 173, 175, 182  
 input data set in cataloged procedure 46  
 JOBLIB 262  
 modify cataloged procedure 55  
 modifying cataloged procedure 54  
 MVS batched compilation 64  
 OPTCD keyword 225  
 parameters for stream I/O 132  
 RECFM keyword 225  
 regional data set 208  
 separate, for index, prime, and overflow areas 172  
 standard data set 64  
 input (SYSIN or SYSCIN) 65  
 output (SYSLIN, SYSPUNCH) 65

ddname  
 %INCLUDE 31  
 list 33  
 standard data sets 64

deblocking of records 103

debugging aids, effect on storage consumption and execution time 305

decimal constants, precision of 338

DECK option 10  
 problems in batched compilation 69

declaration of files 99

declarations, external 332

DECLARE statement 38

DECLARE statements, global optimization of variables 316

defactorization 311

default  
 compile-time option 5

deferred restart 441

define data set 99, 109  
 associating several data sets with one file 102  
 associating several files with one data set 101  
 closing a file 110  
 concatenating several data sets 102  
 ENVIRONMENT attribute 110  
 ESDS 230  
 opening a file 109  
 specifying characteristics 110

define file (*continued*)  
 concatenating several data sets 102  
 ENVIRONMENT attribute 110  
 indexed data set  
 ENV options 169  
 opening a file 109  
 regional data set 188  
 ENV options 188  
 keys 190  
 specifying characteristics 110  
 VSAM data set 218

delete routine, return code 406

delete service routine 406  
 user-defined  
 example of 407

DEN subparameter 108  
 consecutive data set 157

depth of replacement maximum 29

descriptors  
 base elements of structures 384  
 contents 380  
 data types and structures for 380  
 description of 380

DFSORT 348

DFSORT VM 348

diagnostic messages  
 compiler, detecting errors 78  
 preprocessor, detecting errors 78

dimension multiplier 382

direct access  
 indexed data set 181  
 REGIONAL(1) data set 193  
 REGIONAL(2) data set 199  
 REGIONAL(3) data set 205

direct data sets 106, 107

DIRECT file  
 indexed ESDS with VSAM  
 accessing data set 236  
 updating data set 238

RRDS  
 access data set 252

disabled conditions, required processing 344

disabling debugging conditions 344

DISK PLIOPT command option 75

disks for compiler output 74

DISP parameter  
 batch processing 70  
 consecutive data sets 159  
 for consecutive data sets 156  
 for stream I/O 133  
 to delete a data set 123

DISPLAY statement 424

DISPLAY, under VM 89

DO specifications, TO and BY options 336

DO statements  
 expressions in, temporary variables 334

- DO statements (*continued*)
  - repetitive execution of 335
  - special case code 313
- do-groups
  - and storage conservation 334
  - bounds of, computation 334
  - computation of bounds 334
  - evaluating WHILE expressions 334
  - terminating condition of 334
- do-loop, special case code 313
- DSA, size of 331
- DSCB (data set control block) 107, 125
- DSNAME parameter
  - for consecutive data sets 156, 159
  - for indexed data sets 175
  - for retaining data sets 133
  - stream I/O 136
- DSORG subparameter 108
  - indexed data set 174
- dummy records
  - indexed data set 167
  - REGIONAL(1) data set 190
  - REGIONAL(2) data set 196
  - REGIONAL(3) data set 202
  - VSAM 217
- dumps
  - calling PLIDUMP 386
  - defining data sets for
    - DD statement 387
    - logical record length 387
  - identifying beginning of 387
  - PLIDUMP built-in function 386
  - producing Language Environment for MVS & VM
    - dumps 386
    - SNAP 387
- DYNALLOC sort option 353

## E

- E compiler message 44
- E15 input handling routine 361
- E35 output handling routine 364
- EBCDIC (Extended Binary Coded Decimal Interchange Code) 103
- edit-directed transmission
  - matching format lists with data lists 315
- efficient programming
  - assignments and initialization 323
  - condition handling 344
  - data conversion 329
  - data elements 323
  - efficient performance 305, 308
  - expressions and references 326
  - general statements 334
  - global optimization features 308
  - input and output 339

- efficient programming (*continued*)
  - picture specification characters 343
  - program organization 331
  - recognition of names 332
  - record-oriented data transmission 340
  - storage control 332
  - stream-oriented data transmission 341
  - subroutines and functions 338
- elimination of common constants 311
- elimination of common control data 313
- embedded keys 165, 180
- END statement 38
- ENDFILE
  - MVS, entering at terminal 92
  - VM, entering at terminal 88
- entry point
  - sort program 355
- entry point address of module, querying primary 271
- ENTRY statement 38
- entry variable, storage format 374
- entry-sequenced data set
  - defining 231
  - updating 231
  - VSAM 213
    - loading an ESDS 229
    - SEQUENTIAL file 229
    - statements and options 228
- ENVIRONMENT attribute, INDEXED option 340
- ENVIRONMENT options 130, 150
  - ASCII 154
  - BUFFERS 117
  - BUFOFF 155
  - CONSECUTIVE 130, 151
  - CTLASA and CTL360 152
  - D-format and DB-format records 155
  - ENVIRONMENT attribute 110
    - organization options 112
    - other ENVIRONMENT options 113
    - summary table 111
  - equivalent DCB subparameters 113
  - GRAPHIC option 132
  - indexed data set 169
    - ADDBUFF option 169
    - INDEXAREA option 170
    - INDEXED option 170
    - KEYLOC option 170
    - NOWRITE option 172
  - LEAVE|REREAD 153
  - record format options 130
  - RECSIZE 131
  - regional data set 188
  - teleprocessing data set
    - BUFFERS option 258
    - RECSIZE option 258
    - TP option 257
  - TOTAL 151

ENVIRONMENT options (*continued*)

- VSAM
  - BKWD option 220
  - BUFND option 220
  - BUFNI option 220
  - BUFSP option 221
  - GENKEY option 221
  - PASSWORD option 221
  - REUSE option 221
  - SIS option 222
  - SKIP option 222
  - VSAM option 223
- EPLIST 389, 390
  - use of fields 391
- EQUALS sort option 353
- ER-type ESD entry 43
- error conditions, correcting, effect of REORDER option on 317
- errors
  - compiler-detected 71
  - correcting compiler-detected 78
  - correction by compiler 78
  - message severity option 11
  - severity of error compilation 10
- ESD (external symbol dictionary)
  - compile-time option 11
- ESDS (entry-sequenced data set)
  - defining 231
  - updating 231
  - VSAM 213
    - loading 229
    - statements and options 228
- establishing hook exits
  - for code coverage reporting 272—283
  - for function tracing 284
- EVENT option 423
- examining code coverage (example)
  - discussion of programs 272
  - output from 272—273
  - setup for 272
  - source code, CEEBINT 274
  - source code, HOOKUP 280—283
- examples
  - analyzing CPU-time usage 288—304
  - calling PLIDUMP 386
  - examining code coverage 272—283
  - performing function tracing 284
  - relation of data to locators and descriptors 380
  - structure descriptors 385
  - verification program (IEL1MSO1) 444
- exception handler
  - return/reason codes 411
- exception router
  - return/reason codes 410
- exception router service routine
  - preinitialized program 409
- EXCLUSIVE files, noncompatibility with VSAM 224
- EXEC statement
  - cataloged procedure, modifying 54
  - compiler 64
  - introduction 64
  - maximum length of option list 68
  - minimum region size 64
  - modify cataloged procedure 54
  - MVS batched compilation 55, 64
  - PARM parameter 68
  - to specify options 68
- EXECUTE extended parameter list request 390
- execution
  - suppressing in the case of error 79
  - VSAM 211
- Exit (E15) input handling routine 361
- Exit (E35) output handling routine 364
- expressions
  - common, effect of scope on optimization 319
  - form of, inhibiting optimization 319
  - optimization of
    - base register allocation for branch instructions 311
    - branch instructions 311
    - common constants 311
    - common expression elimination 309, 318, 319, 320
    - constant exponents 311
    - constant expressions 311
    - constant multipliers 311
    - constants in array subscripts 311
    - defactorization 311
    - elimination of common constants 311
    - inhibiting common expression elimination 318, 319, 320
    - modification of loop control variables 310
    - redundant expression elimination 310
    - replacement of constant exponents 311
    - replacement of constant expressions 311
    - replacement of constant multipliers 311
    - simplification of expressions 310, 311
  - precision of variables in 324
  - scale factor of variables in 324
  - transferring constant outside of loops 312
- expressions, for performance improvement 326
- extended binary coded decimal interchange code (EBCDIC) 103
- extended parameter list 389, 390
  - use of fields 391
  - example of 394
- EXTERNAL attribute 37
- external declarations 332
- external entries information control block
  - layout of 270
  - specifying pointer to 268

- external procedures, designing and writing 331
- external references
  - concatenation of names 100
  - ESD entry 41
- external symbol dictionary (ESD)
  - compiler listing 41
  - ESD entries 42

## F

- F option of ENVIRONMENT
  - for record I/O 113
  - for stream I/O 130
- F-format records 104
- FB option of ENVIRONMENT
  - for record I/O 113
  - for stream I/O 130
- FB-format records 104
- FBS option of ENVIRONMENT
  - for record I/O 113
  - for stream I/O 130
- FBS-format records 104
- field for sort 352
- FILE attribute 37
- file variable, storage format 374
- files
  - associating data sets with files 99
  - closing 110
  - defining data sets 99
  - establishing characteristics 102
  - share between tasks 425
  - TRANSIENT 107, 256
  - used by the compiler 74
- FILLERS tab set table field 140
- FILSZ sort option 353
- FINISH condition, avoiding the use of ON-units 344
- fixed-length records 104
- fixed-length records, record format 339
- FLAG option 11
- flowchart for sort 362
- FMARGINS
  - input record option 28
- format items, termination of processing 342
- format lists
  - contained in FORMAT statements 341
  - matched with data lists in edit-directed transmission 315
- format notation, rules for xviii
- format of PLIOPT command 76
- FORMAT statements, containing format list 341
- FORTRAN
  - map structure 39
- free-storage routine, return code 409
- free-storage service routine
  - preinitialized program 409

- FS option of ENVIRONMENT
  - for record I/O 113
  - for stream I/O 130
- FS-format records 104
- FSEQUENCE 21
- fullword subscript compatibility 8
- FUNC subparameter 108
  - consecutive data set 157
- function reference and passing parameters 373
- function tracing 272
- functions for performance improvement 338

## G

- generating an Language Environment for MVS & VM
  - dump using PLIDUMP 386, 388
- GENKEY option 111, 118, 221
  - VSAM 219
- get-storage routine
  - preinitialized program 408
  - return code 408
- global optimization of variables 316
- GONUMBER option 11
  - storage requirements for 306
- GOSTMT option
  - compile-time option 12
  - storage requirements for 306
- GOTO statements, referencing label variables 334
- graphic data 129
- GRAPHIC option 132
  - compile-time option 12
  - ENVIRONMENT option 111
  - stream I/O 130
- graphic string constant compilation 12

## H

- handling routines
  - data for sort
    - input (sort exit E15) 361
    - output (sort exit E35) 364
  - PLISRTB 366
  - PLISRTC 367
  - PLISRTD 367
  - to determine success 358
  - variable length records 369
- header label 107
- heading information for compiler list 34
- hexadecimal
  - address representation 42
- HIR\_BLOCK (parameter of IBMBHIR) 271
- HIR\_EPA (parameter of IBMBHIR) 271
- HIR\_LANG\_CODE (parameter of IBMBHIR) 271
- HIR\_NAME\_ADDR (parameter of IBMBHIR) 271
- HIR\_NAME\_LEN (parameter of IBMBHIR) 271

- HIR\_PATH\_CODE (parameter of IBMBHIR) 271
- hook exits
  - establishing to perform function tracing 284
  - establishing to report on code coverage 272—283
  - using 272
- hook information
  - control block
    - layout of 270
    - specifying pointer to 268
  - obtaining 271
  - retrieval module 271
  - using IBMBHIR 271
- hook services
  - activating hooks 265—266
  - IBMBHIR 271
  - IBMBHKS 265—266
  - IBMBSIR 266
  - IBMHSIR 271
  - obtaining hook information 271
  - obtaining static information on compiled modules 266
  - purpose of activating 265
  - supported environments 265
- hooks
  - activating 265
  - location suboptions 26
  - querying %PATHCODE value 271
  - retrieving information about 271
- HOOKUP (sample program) 272
  - output from 273, 289
  - source code for 280—283, 297—304
- HOOKUPT (sample program) 272
  - output from 284
  - source code for 287

## I

- I compiler message 44
- IBMBHIR
  - discussion of 271
  - HIR\_BLOCK (parameter) 271
  - HIR\_EPA (parameter) 271
  - HIR\_LANG\_CODE (parameter) 271
  - HIR\_NAME\_ADDR (parameter) 271
  - HIR\_NAME\_LEN (parameter) 271
  - HIR\_PATH\_CODE (parameter) 271
  - primary entry point address of module 271
  - programming language used to compile module 271
  - use of 271
- IBMBHKS
  - declaring 265
  - discussion of 265—266
  - examples of use 274—279
  - function codes 266
  - instead of debugging tool 265

- IBMBHKS (*continued*)
  - invoking 265
  - programming interface 265, 266
  - return codes 266
  - using 265, 266
  - warning about 266
- IBMBSIR
  - block information control block
    - layout of 270
    - specifying pointer to 268
  - control block elements 267
  - discussion of 266
  - external entries information control block
    - specifying pointer to 268
  - external entries information control block, layout of 270
  - function codes for 267—268
  - hook information control block
    - layout of 270
    - specifying pointer to 268
  - invoking 267
  - module information control block
    - layout of 269
    - specifying pointer to 268
  - programming interface 267
  - return codes for 268
  - SIR\_A\_DATA (parameter) 268
  - SIR\_ENTRY (parameter) 268
  - SIR\_FNCCODE (parameter) 267
  - SIR\_MOD\_DATA (parameter) 268
  - SIR\_RETCODE (parameter) 268
  - specifying main entry point for 268
  - specifying pointers
    - block information control block 268
    - external information control block 268
    - hook information control block 268
    - module information control block 268
  - specifying type of static information 267
  - uses of 266
- IBMHSIR
  - invoking 271
  - programming interface 271
  - returned information
    - about blocks 271
    - about hooks 271
    - about modules 271
- ID ESD heading 42
- identifiers
  - not referenced 8
  - source program 8
- IEL1C cataloged procedure 47
- IEL1CG cataloged procedure 51
- IEL1CL cataloged procedure 48
- IEL1CLG cataloged procedure 50
- IEL1MSO1 (sample program) 444

IF statement  
   branching optimized 311  
   improving efficiency by compound expression 310  
 imprecise interrupt localization 12  
 IMPRECISE option 12  
 in-line code  
   for built-in functions 314  
   for conversions 314  
   for record I/O 314  
   for string manipulation 314  
   string built-in functions 339  
 %INCLUDE  
   allocating data sets 63  
   compiler 30  
   source statement library 67  
   TSO 62, 63  
   VM 75, 77  
   without full preprocessor 13  
 INCLUDE option 13  
 independent overflow area 167, 177  
 index  
   cylinder 166  
   master 166  
   track 166  
   upgrade 212  
 index area 166  
 INDEXAREA option 111, 170  
 INDEXED data set, direct update of 340  
 indexed data sets  
   accessing and updating 179  
   creating 172, 179  
   DD statement 174  
   defining files for 169  
   direct access 181  
   dummy records 167  
   index area separate DD statement 172  
   index structure 166  
   indexed sequential data set 106  
   indexes 166  
   master index 178  
   name of 175  
   organization 163  
   overflow area 177  
   record format and keys 175  
   reorganizing 184  
   REWRITE statement 165  
   sequential access 180  
   SEQUENTIAL files 165  
   specifying ENVIRONMENT options 169  
   SYSOUT device restriction 175  
   updating 183  
   using indexes 166  
   using keys 163  
 indexed ESDS (entry-sequenced data set)  
   alternate indexes 239  
   DIRECT file 236  
   loading 234  
   SEQUENTIAL file 236  
 INDEXED file with VSAM 224, 226  
 INDEXED option 111, 170  
 information interchange codes 103  
 inhibiting optimization  
   accessing array elements 319  
   common expression elimination 318, 319, 320  
   condition handling 320  
   for loops 313  
   form of expressions 319  
   limit on global optimization of variables 316  
   on variables 316  
   ORDER option 316  
   REORDER option 316  
   scope of common expressions 319  
   using ORDER option 313, 315  
 inhibiting reordering, using ORDER option 313  
 INIT extended parameter list request 389  
 INITIAL attribute 38  
 INITIAL attribute, array restriction 333  
 INITIAL attribute, on external noncontrolled variable 333  
 initial volume label 107  
 initialization  
   arrays 313  
   for performance improvement 323  
   of arrays and structures 313  
   propagating values 313  
   structures 313  
 input  
   compiler  
     data sets 65  
     input record limit 16  
     record format 28  
   data for sort 360  
     PLISRTA 365  
   defining data sets for stream files 130  
   in cataloged procedures 47  
   MVS, punctuating long lines 91  
   performance improvement 339  
   routines for sort program 360  
   SEQUENTIAL 156  
   skeletal code for sort 363, 364  
   sort data set 359  
   specify input record section 21  
   VM, punctuating long lines 87  
 input data, compile and link-edit 48  
 input/output  
   inhibiting optimization 318  
   optimization, inhibiting 318  
 INSOURCE option 13  
 insufficient storage 22  
 interactive program 146  
   attention interrupt 13

- interblock gap (IBG) 103
- interchange codes 103
- interface
  - preinitialized program 390
- INTERNAL attribute 37
- internal switches and counters, FIXED BINARY data type 324
- INTERRUPT option 13
- interrupt, debugging tool 437
- interruptions
  - attention interrupts under interactive system 13
  - ATTENTION ON-units 437
  - debugging tool 437
  - imprecise interruptions localization 12
  - main description 436
  - passing control to PLITEST 437
- invoke
  - cataloged procedures 52
    - link-editing multitasking programs 53, 54
    - multiple invocations 52
  - preprocessor 29
- invoking an alternative MAIN 398
- ISAM data set considerations 211

## J

- JCL
  - batched processing 70
  - for the compiler 67
  - improving efficiency 46
  - reducing errors 46
- JOBLIB DD statement 262

## K

- KEY condition on LOCATE statements 341
- key handling for REGIONAL data sets 314
- key indexed VSAM data set 216
- key-sequenced data sets
  - accessing with a DIRECT file 236
  - accessing with a SEQUENTIAL file 236
  - alternate indexes for 239
  - loading 234
  - statements and options for 232
- KEYLEN subparameter 108
  - indexed data set 174
- KEYLENGTH option 111, 121
  - comparison with DCB subparameter 113
  - sequential access for indexed data sets 180
- KEYLOC option 111, 170
  - comparison with DCB subparameter 113
- KEYLOC value 171
  - effect on embedded keys 171
  - indexed data set 170, 174, 177
- keys
  - indexed data set 163

- keys (*continued*)
  - optimizing handling for REGIONAL data sets 314
  - REGIONAL(1) data set 190
    - dummy records 190
  - REGIONAL(2) and (3) data sets 195
    - dummy records 196
  - VSAM
    - indexed data set 216
    - relative byte address 216
    - relative record number 216
- KEYTO option
  - REGIONAL (2) data set 199
  - REGIONAL (3) data set 204
  - under VSAM 229
- KSDS (key-sequenced data set)
  - define and load 235
  - unique key
    - ESDS 240
  - update 237
  - VSAM
    - alternate indexes 239
    - DIRECT file 236
    - loading 234
    - methods of insertion 239
    - SEQUENTIAL file 236

## L

- LABEL attribute, restriction on INITIAL attribute 333
- label constant, storage format of 374
- LABEL parameter 133
  - for magnetic tape 157
  - for stream I/O 133
  - stream I/O 136
- label register ESD entry 42
- label variables
  - effect on optimization of loops 313
  - referenced in GOTO statements 334
  - storage format of 374
- labeling volumes 107
- labels for data sets 107
- labels, standard, and VM 84
- LANGLVL compile-time option
  - NOSPROG suboption 14
  - SPROG suboption 14
- Language Environment for MVS & VM library xv
- LD-type ESD entry 43
- LEAVE and REREAD options 154
- LEAVE option 111, 153
  - for stream I/O 130
- LEAVE statement 38
- LENGTH
  - ESD heading 42
  - RECORD option for sort 354
- LIB(dslist) command 61

- libraries
  - compiled object modules 126
  - creating a data set library 124
  - creating a member 127
  - creating and updating a library member 125
  - creating, examples of 125
  - directory 124
  - extracting information from a library directory 128
  - general description of 107
  - how to use 123
  - information required to create 124
  - multitasking with cataloged procedure 53
  - placing a load module 126
  - source statement library 55, 67
  - SPACE parameter 124
  - structure 128
  - system procedure (SYS1.PROCLIB) 123
  - types of 123
  - updating a library member 127
  - using 123
- library calls, in-line code substitution 314
- library routines
  - Language Environment for MVS & VM 315
  - PL/I for MVS & VM 315
  - run-time 315
- library stubs 315
- LIMCT subparameter 108, 209
- limit on global optimization of variables 316
- line length 139
- line numbers in messages 11
- LINE option 131, 139
- LINECOUNT option 14
- lines in compiler list 14
- LINESIZE option
  - OPEN statement 131
  - tab set table field 140
- LINK macro instruction 32
- link-edit, selecting math results 80
- link-editing, description of 80
- linkage editor
  - suppress link-editing 79
- LIST
  - compile-time option 14
- listing
  - cataloged procedures 52
  - check out program listings 78
  - compiler 62
  - compiler listing 33
    - aggregate length table 38
    - ATTRIBUTE and cross-reference table 36
    - attribute table 37
    - cross-reference table 37
    - ddname list 5, 33
    - ESD entries 42
    - external symbol dictionary 41
    - heading information 34
    - messages 44
  - listing (*continued*)
    - compiler listing (*continued*)
      - object listing 44
      - options 34
      - preprocessor input 34
      - return codes 45
      - SOURCE program 34
      - statement nesting level 36
      - statement offset addresses 40
      - static internal storage map 43
      - storage requirements 39
      - TSO 55, 62
    - MVS batched compilation 55, 67
    - object module 44
    - source program 23
    - statement offset address 40
    - static internal control section 44
    - SYSPRINT 67
  - LMESSAGE option 15
  - load module
    - to name 70
    - to substitute 17
  - load service routine
    - example of 404
    - parameters passed to 403
  - loader program, using 51
  - local optimization of variables 316
  - localization of imprecise interrupts 12
  - LOCATE statements, KEY condition on 341
  - locator
    - contents 380
    - data types and structures for 380
  - locator/descriptors
    - contents 380
    - data types and structures for 380
    - description of 380
  - locators
    - aggregate locators
      - array descriptor addresses 381
      - array starting addresses 381
      - contents of 381
      - structure descriptor addresses 381
      - structure starting addresses 381
    - area locator/descriptor 381
    - array descriptors
      - bounds components 383
      - concatenation of string or area descriptor with 382
      - contents of 382
      - multiplier components 383
      - relative virtual origin component 383
    - arrays of structures 385
    - description of 380
    - string locator/descriptor 383
    - structure descriptors 384
    - structures of arrays 385

- logical not 18
- logical or 21
- loop control variables, modification of 310, 312
- loops
  - constant expressions, transferring 312
  - constant statements, transferring 312
  - effect of REORDER option on 316
  - label variables and optimization 313
  - optimization of
    - effect of label variables 313
    - inhibiting 313
    - maintaining control values in registers 313
    - modification of loop control variables 312
    - transfer of expressions from 312
    - transferring constant expressions 312
    - transferring constant statements 312
    - unrecognizable 313
  - optimization, modification of control variable 310, 312
  - recognition of optimization purposes, transfer of expressions 312
  - special case code 313
  - transfer of expressions from 312
  - transferring constant expressions or statements 312
  - undesired effect of optimization 312
  - unrecognizable for optimization 313
  - use of registers for modified values 315
- LRECL subparameter 103, 108
  - consecutive data set 157
  - indexed data set 174

## M

- MACRO option 15
- magnetic tape 137, 150
  - LABEL parameter 157
- MAIN procedure parameter list, format of 376
- MAIN procedure parameters, passing 376
- main storage for sort 355
- MAP option 15
- MARGINI option 16
- MARGINS
  - compile-time option
    - and input records 16
    - variable records 16
- MARGINS option 16
- mass sequential insert 238
- master index 166, 178
- matching format lists with data lists 315
- math results, selecting at link-edit 80
- maximum
  - block-size 116
  - record length 115
  - sort record length 354
- MDECK option 17
  - problems in batched compilation 69

- message
  - check out program messages 78
  - compiler error severity option 11
  - compiler list 44
  - control program (MCP) 255
    - printed format 142
    - processing (TCAM MMP) 256, 262
    - run-time message line numbers 11
    - statement numbers in run-time messages 12
    - to specify length 15
- message router
  - return code 416
  - service routine
    - example of 417
    - preinitialized program 416
- minimizing paging 307, 308
- minimum region size
- mixed string constant compilation 12
- MODE subparameter 108
  - consecutive data set 157
- modify
  - cataloged procedures 54
    - DD statement 55
    - EXEC statement 54
- modifying loop control variables 310, 312
- modular programming
  - advantages of 331
  - and optimization 331
- module information control block
  - layout of 269
  - specifying pointer to 268
- modules
  - create and store object module 19
  - name a load module during compilation 70
  - object module identification code 10
  - querying
    - address of name 271
    - length of name 271
    - programming language used to compile 271
    - retrieving information about 266
    - substitute file name 17
- moving expressions out of loops
  - undesired effect
- multiple
  - procedures 69
- multiple independent
  - computations 432
    - nontasking 433
    - tasking 434
  - processes 428
    - nontasking 429
    - tasking 430
- multiple invocations
  - cataloged procedures 52
  - cataloged procedures, environment 52
  - preinitialized program 389

- multiple invocations (*continued*)
  - run-time environment
    - CALL extended parameter list request 389
    - description of 389
    - EXECUTE extended parameter list request 390
    - INIT extended parameter list request 389
    - preinitializable program 389
    - preinitializable programs extended 389
    - TERM extended parameter list request 390
- multiplication operations
  - loop optimization 310, 312
  - optimization by repeated addition 310, 312
- MULTIPLY built-in function 339
- multitasking
  - create tasks 423
  - independent computations 432
  - independent processes 428
  - library (SYS1.SIBMTASK) 53
  - options in PLIDUMP 386
  - performance improvement 345
  - priority 427
  - reliability 426
  - running 428
  - sharing data 425
  - sharing files 425
  - synchronization and coordination 424
  - tasking facilities 422
  - terminating 426
- MVS
  - attention router 412, 413
    - example of 413
  - batch compilation 64
    - batched processing JCL 70
    - compiler JCL 67
    - DD statement 64
    - examples of 71
    - EXEC statement 64, 68
    - listing (SYSPRINT) 67
    - multiple procedures/single job step 69
    - NAME option 69
    - return codes 70
    - SIZE option 69
    - source statement library (SYSLIB) 67
    - specifying options 68
    - temporary workfile (SYSUT1) 66
  - calling preinitialized program under MVS 419
  - delete service routine 406, 407
    - example of 407
  - exception router service routine 409
  - free-storage service routine 409
  - general compilation 55, 56
  - get-storage routine 408
  - interface for 390
  - invoking an alternative MAIN 398
  - load service routine 403, 404
    - example of 404

- MVS (*continued*)
  - message router service routine 416
    - example of 417
  - minimum region size
  - preinitialized program 419
  - using service vector 402
  - using, example of 398
- MVS integrity 390
- MVS/XA, compiler 71

## N

- NAME
  - compile-time option 17
  - MVS compilation 69
- name indexed data set 175
- names recognition 332
- NCP subparameter 108, 119
  - comparison with DCB subparameter 113
  - ENVIRONMENT option 111
- negative value
  - block-size 116
  - record length 115
- NEST option 18
- nested blocks, declaring arithmetic variables 316
- NOEQUALS sort option 353
- NOINTERRUPT 13
- NOMAP option 39
- NONE
  - hooks location suboption 26
- NOPRINT
  - noprint 32
  - NOPRINT command 61
  - PLIOPT command option 75
- NOSPROG suboption of LANGLVL 14
- NOSYNTAX option 24
- NOT option 18
- note statement 44
- NOWRITE option 172
  - ENVIRONMENT option 111
- NTM subparameter 108
  - creating a master index 178
  - indexed data set 174
- null statements, replacing IF statements 335
- NUMBER option 18

## O

- object
  - compatibility 8
  - listing 44
  - module
    - create and store 19
    - identification code 10
    - record size 65
  - OBJECT option 19
    - batched compilation 69

- object (*continued*)
  - OBJECT option (*continued*)
    - PLIOPT command 75
- object code
  - and self-defining data 332
  - for procedures, size of 332
- object program, library stubs 315
- obtaining denser packing of data 325
- obtaining hook information on compiled modules 271
- obtaining static information on compiled modules 266
- offset
  - address list 19
  - OFFSET compile-time option 19
  - offset of tab count 140
  - table 40
- ON statements, execution order 344
- ON-units
  - effect of ORDER option on 316, 320
  - effect of REORDER option on 317
  - inhibiting optimization 318
  - optimization, inhibiting 318
  - priority of operands 320
  - recommended use of 345
- OPEN statement 109
- OPEN statements, specifying more than one file 340
- OPTCD subparameter 108
  - consecutive data set 157
  - DD statement 225
  - indexed data set 174
  - overflow area 177
- optimization
  - arrays 322
  - do-loops 322
  - global on variables 316
  - in-line code 322
  - in-line conversions 322
  - inhibiting
    - accessing array elements 319
    - common expression elimination 318, 319, 320
    - condition handling 320
    - for loops 313
    - form of expressions 319
    - limit on global optimization of variables 316
    - on variables 316
    - ORDER option 316
    - REORDER option 316
    - scope of common expressions 319
    - using ORDER option 313, 315
  - limitation on flow analysis 331
  - local on variables 316
  - loops, maintaining control values in registers 313
  - overview of types 308
  - register allocation and addressing 322
  - structures 322
  - types of 315
    - array assignments 313
    - base register allocation for branch instructions 311
- optimization (*continued*)
  - types of (*continued*)
    - branch instructions 311
    - common constants 311
    - common expression elimination 309, 318, 319, 320
    - constant exponents 311
    - constant expressions 311
    - constant expressions in loops 312
    - constant multipliers 311
    - constant statements in loops 312
    - constants in array subscript expressions 311
    - data lists matched with format lists 315
    - defactorization 311
    - elimination of common constants 311
    - for expressions 309, 312
    - format lists matched with data lists 315
    - in-line code for built-in functions 314
    - in-line code for conversions 314
    - in-line code for record I/O 314
    - in-line code for string manipulation 314
    - initialization of arrays 313
    - initialization of structures 313
    - key handling for REGIONAL data sets 314
    - matching format lists with data lists 315
    - modification of loop control variables 310, 312
    - redundant expression elimination 310
    - register allocation 315
    - register usage for loops 315
    - replacement of constant exponents 311
    - replacement of constant expressions 311
    - replacement of constant multipliers 311
    - simplification of expressions 310, 311
    - structure assignments 313
    - transfer of expressions from loops 312
    - transferring constant expressions or statements outside of loops 312
- optimization features
  - global 315
    - common expression elimination 318, 319, 320
    - condition handling 320
    - ORDER and REORDER options
    - ORDER option 316
    - REORDER option 316
    - transfer of invariant expressions 321
    - variables 316
  - optimized code 322
  - other optimization features 322
  - redundant expression elimination 322
- OPTIMIZE option 20
  - optimization features of 308, 315
  - optimization features of, for expressions 309, 312
- optimized code 322
- optimized, format list with data list
- options
  - for compilation 34

- options (*continued*)
  - for creating regional data set 186
  - indexed data sets 164
  - option list
    - address parameter 32
    - compiler 33
    - PLI command 59
  - reinstate options deleted from installation 10
  - to specify for compilation 68
- OPTIONS option 20
- OPTIONS(ASSEMBLER) attribute, and argument list
  - addresses 374
- OPTIONS(MAIN) attribute 332
- OR option 21
- ORDER option 321
  - effect on begin-blocks 316
  - effect on ON-units 316, 320
  - effect on procedures 316
  - inhibiting loop optimization 313
  - inhibiting optimization 316
  - inhibits loop optimization
    - optimization and register allocation 315
    - when to specify 316
- organization
  - indexed data set 163
  - modular programming 331
  - program performance improvement 331
  - teleprocessing 256
- OS-simulated data set 82
- output
  - data for sort 360
    - PLISRTA 365
  - defining data sets for stream files 130
  - from the compiler 73
  - limit preprocessor output 17
  - performance improvement 339
  - punched card 10
  - routines for sort program 360
  - SEQUENTIAL 156
  - skeletal code for sort 364
  - sort data set 359
  - SYSLIN 65
  - SYSPUNCH 65
- output files, blank after last value 343
- overflow area 167
  - indexed data set 177
  - main discussion of 177
  - separate DD statement 172
- overlay defining 342

## P

- page
  - ENVIRONMENT option 131
  - page 32
  - page number
    - compiler list 33

- page (*continued*)
  - PAGELength tab set table field 140
  - PAGESIZE tab set table field 140
- paging
  - items accessed together 307
  - minimizing 307, 308
  - read-only pages 307
- pairing alternate index path with a file 211
- parameters, passing 373
- PARM parameter 54
  - specify options 68
- passing arguments 373
- passing MAIN procedure parameters 376
- passing parameters 373
- password
  - PASSWORD option 221
- PATH
  - hooks location suboption 26
- pending condition 256
- % statements 32
- performance improvement
  - tuning a program 305, 306, 308
  - virtual storage system 307, 308
    - assigning storage classes for based and controlled variables 307
    - avoiding large branches in source code 307
    - controlling the positioning for variables 308
    - designing and programming modular programs 307
    - handling aggregates larger than page size 307
    - making static internal CSECT read-only 308
    - minimizing paging 307
    - placing CSECTs within pages 308
    - placing variables within CSECTs 308
  - VSAM options 223
- performing function tracing (example)
  - discussion of programs 284
  - output from 284
  - setup for 284
  - source code for 284—287
- picture specification characters 343
  - allowable for in-line arithmetic data conversion 329
- PICTURE specifications
  - checking picture data 344
  - point picture character 343
  - scale factor 343
  - V character 343
- PL/I for MVS & VM library xiv
- PLI command
  - data set name 59
  - LIB(dslist) 61
  - main discussion of 59
  - NOPRINT 61
  - option list 59
  - PRINT 60
  - PRINT(\*) 60

- PLI command (*continued*)
  - SYSPRINT 61
  - TSO 55, 59
- PLICANC statement, and checkpoint/request 441
- PLICKPT built-in subroutine 438
  - arguments 439
  - requesting a checkpoint record 439
- PLIDUMP
  - calling to produce an Language Environment for MVS & VM dump 386
  - converting to CEE3DMP options
  - example of 386
  - H option 387
  - options 386
  - syntax of 386
  - user-identifier 387
- PLIOPT command
  - format 76
  - options 75
    - %INCLUDE statement 75
    - %INCLUDE under VM 77
  - format 76
  - TXTLIB 77
  - VM compilation to run under MVS 77
  - special requirements 77
  - use 73
    - batched compilation 78
    - compile-time options 74
    - compiler output 73
    - files 74
    - PLIOPT command options 75
- PLIREST statement 441
- PLIRETC built-in subroutine
  - return codes for sort 358
- PLISRTA 365
  - example of 365
- PLISRTB 366
  - example of 366
- PLISRTC 367
  - example of 367
- PLISRTD 367
  - example of 368
- PLISRTx 355
  - arguments 349
  - entry points 348, 355
- PLITABS
  - control section 141, 142
  - declaration 89
- PLITEST
  - attention processing 437
  - to specify available capabilities 26
- PLIXOPT variable, use in tuning 306
- POINTERADD 14
- pointers
  - passing 375
  - set in READ SET or LOCATE 340
- pointers (*continued*)
  - use in expressions 14
  - validity of setting 340
- POINTERVALUE 14
- precision of decimal constants 338
- preinitialized program
  - attention router 412, 413
    - example of 413
  - calling preinitialized program under MVS 419
  - calling preinitialized program under VM 419
  - delete service routine 406, 407
    - example of 407
  - exception router service routine 409
  - free-storage service routine 409
  - get-storage routine 408
  - interface for 390
  - invoking an alternative MAIN 398
  - load service routine 403, 404
    - example of 404
  - message router service routine 416
    - example of 417
  - MVS 419
  - preinitializing a program 393, 394
  - SYSTEM option 419
  - user exit 419
  - using service vector 402
  - using the preinitialized program 398
  - using, example of 398
  - VM 419
- preinitialized program interface
  - extended parameter list 391
    - use of fields 391
- preparation for sort 348
  - determining storage 355
  - sorting field 352
  - specify records 354
  - type of sort 349
- preprocessor
  - %INCLUDE statement 30
  - description of 29
  - discussion of 29
  - for program testing 31
  - input 34
  - invoking 29
  - limit output to 80 bytes 17
  - output format 29
  - program testing 31
  - source program 15
  - with MACRO 15
- primary entry point address of module, querying 271
- prime data area 167
  - separate DD statement 172
- print
  - PLIOPT command option 75
  - print 32
  - PRINT file
    - format 145

print (*continued*)  
 PRINT file (*continued*)  
   line length 139  
   stream I/O 138  
 PRINT files  
   changing format, input 86  
   formatting conventions 89  
   punctuating output 89  
     PRINT(\*) command 60  
   printer control character 29  
   record I/O 162  
 priority  
   apparent and actual 427  
   tasks 427  
 priority of operands, effect on ON-units 320  
 PRIORITY option 424  
 procedures  
   cataloged, using under MVS 46  
   compile and link-edit (IEL1CL) 48  
   compile only (IEL1C) 47  
   compile, link-edit, and run (IEL1CLG) 50  
   compile, load and run (IEL1CG) 51  
   containing more than one entry point 338  
   effect of ORDER option on 316  
   entry points, containing more than one 338  
   external, designing and writing 331  
   given initial control 332  
   object code, size of 332  
   PROCEDURE statement 38  
 PROCESS statement 28  
   override option defaults 68  
 processing jobs in the background 63  
 program branch code 311  
 program control  
   control section 42  
 program organization 331  
 program testing, using preprocessor 31  
 programming interfaces  
   IBMBHKS 265  
   IBMBSIR 267  
   IBMHSIR 271  
 programming language used to compile module,  
 querying 271  
 programs 284, 288  
   conversational, creating 89  
   designing and writing 331  
   external procedure for 331  
   modular programming 331  
   size of 331  
   tuning  
     for virtual storage system 307, 308  
     source code 305  
 prompting  
   automatic, overriding 90  
   automatic, using 90

protection exception, avoiding 325  
 PRTSP subparameter 108  
   consecutive data set 157  
 pseudoregister ESD entry 42  
 punctuating  
   MVS  
     continuation character 91  
     GET DATA statement 91  
     GET LIST statement 91  
   VM  
     continuation character 87  
     GET DATA statement 88  
     GET LIST statement 88  
 punctuation  
   automatic prompting  
     overriding 90  
     using 90  
   MVS  
     automatic padding for GET EDIT 91  
     entering ENDFILE at terminal 92  
     long input lines 91  
     SKIP 92  
   output from PRINT files 89  
   VM  
     automatic padding for GET EDIT 88  
     entering ENDFILE at terminal 88  
     long input lines 87  
     SKIP 88  
 PUT DATA statement, using without a data list 343  
 PUT EDIT command 146

## Q

queues 255

## R

REAL attribute 37  
 RECFM subparameter 108  
   compatibility interface 225  
   consecutive data set 157  
   indexed data set 174  
 record  
   checkpoint 439  
   data set 440  
   deblocking 103  
   maximum size for compiler input 65  
   sort program 354  
   specify compiler input record limit 16  
   specify input record section 21  
 record format 103, 137, 175  
   fixed-length records 104  
   indexed data set 175, 176, 177  
   options 130  
   to specify 150  
   undefined-length records 106

- record format (*continued*)
  - variable-length records 104
- record I/O 113, 149
  - data set
    - access 157
    - consecutive data sets 159
    - create 156
    - types of 121
  - ENVIRONMENT option 150
  - file
    - define 150
  - in-line code for 314
  - magnetic tape without standard labels 137, 150
  - performance improvement 340
  - record format 150
- record length 115
  - indexed data set 173
  - regional data sets 185
  - specify 103
  - variable 177
- RECORD statement 354
- recorded key 190
  - indexed data sets 163
  - KEYTO option 204
- RECSIZE option 114, 258
  - comparison with DCB subparameter 113
  - consecutive data set 131
  - defaults 131
  - for stream I/O 130—131
  - syntax 114
- recursive assembler routine
  - RECURSIVE attribute 332
- reduce storage requirement 20
- redundant expression
  - definition of 310
  - elimination of 310
  - example of 310
- redundant expression elimination 310, 322
- REFER option (self-defining data) 332
- references for performance improvement 326
- referencing functions and passing parameters 373
- region
  - background region for TSO 55, 63
  - REGION parameter 54
  - region size
    - EXEC statement 64
    - minimum
- REGION size, minimum required 46
- regional data sets
  - avoiding conversion of source keys 340
  - DD statement
    - accessing 210
    - creating 209
  - defining files for 188
    - specifying ENVIRONMENT options 188
    - using keys 190
- regional data sets (*continued*)
  - key handling optimization
    - for REGIONAL(1) 314
    - for REGIONAL(2) 314
    - for REGIONAL(3) 314
  - key handling optimized
  - operating system requirement 208
  - REGIONAL(1) data set 190
    - accessing and updating 192
    - creating 191
    - using 190
  - REGIONAL(2) data set 195
    - accessing and updating 198
    - creating 197
    - keys for 195
    - using 195
  - REGIONAL(3) data set 202
    - accessing and updating 204
    - creating 202
    - keys for 195
    - using 202
  - source keys, avoiding conversion of 340
  - VSAM 226
- REGIONAL option 188
- register usage for loops 315
- registers, allocating, effect of REORDER option 315, 317
- relative byte address (RBA) 216
- relative record number 216
- relative virtual origin (RVO) of arrays 382
- relative-record data sets
  - accessing with a DIRECT file 252
  - accessing with a SEQUENTIAL file 251
  - loading 249
  - statements and options for 247
- reliability of tasking programs 426
- REORDER option
  - effect in correcting error conditions 317
  - effect on loops 316
  - effect on ON-units 317
  - effect on register allocation 315, 317
  - for loop optimization 312
  - inhibiting optimization 316
  - when to specify 316
- reordering, inhibiting using ORDER option 313
- reorganizing indexed data set 184
- REPEAT option
  - in-line code for 314
- replacement maximum 29
- replacement of constant expressions 311
- replacement of constant multipliers and exponents 311
- REPLY, under VM 89
- reporting on CPU-time usage (example) 288—304
- REREAD option of ENVIRONMENT 111
  - for stream I/O 130

- restart
  - RESTART parameter 441
  - to request 440
    - automatic after system failure 440
    - automatic within a program 441
    - deferred restart 441
    - to cancel 440
    - to modify 441
- retaining environment for multiple invocations
  - preinitialized program 389
- retrieving hook information using IBMHIR 271
- retrieving information on compiled modules 265
- retrieving static information using IBMBSIR 266
- return code
  - checkpoint/restart routine 439
  - PLIRETC 358
- return codes
  - attention handler 412
  - attention router 412
  - batched compilation 70
  - delete routine 406
  - exception handler 411
  - exception router 410
  - free-storage routine 409
  - get-storage routine 408
  - in compiler listing 45
  - message router 416
- REUSE option 111, 221
- rewriting records contained in buffers 340
- RKP subparameter 108, 171
  - effect on embedded keys 171
  - indexed data set 170, 174, 177
- RPTSTG run-time option 306
- RRDS (relative record data set)
  - define 250
  - load statements and options 247
  - load with VSAM 249
  - update 254
  - VSAM
    - DIRECT file 252
    - loading 249
    - SEQUENTIAL file 251
- run-time
  - message line numbers 11
- run-time considerations
  - VM
    - automatic prompting 86
    - changing PRINT file format 86
    - formatting conventions 86
    - input restrictions 86
    - OS data sets 84
    - output restrictions 86
    - PL/I conventions 86
    - record I/O at terminal 85
    - restrictions 85
    - separately compiled procedures 81
    - stream I/O conventions 86

- run-time considerations (*continued*)
  - VM (*continued*)
    - using data sets and files 81
- run-time library routines 315
- run-time options
  - RPTSTG 306
  - STACK 306
- run-time tuning 265, 304
- run-time, VM considerations 80
- running
  - tasking programs 428

## S

- S compiler message 44
- SAMEKEY built-in function 242
- sample program IEL1MSO1 444
- SBCS-line continuation character 87
- SCALARVARYING option 121
- SCALARVARYING option of ENVIRONMENT 219
- section definition ESD entry 41
- self-defining data (REFER option) 332
- sequence number 21
- SEQUENCE option 21
- sequential access
  - REGIONAL(1) data set 193
  - REGIONAL(2) data set 199
  - REGIONAL(3) data set 204
- sequential data set 106
- SEQUENTIAL file
  - ESDS with VSAM
    - defining and loading 230
    - updating 231
  - indexed ESDS with VSAM
    - access data set 236
  - RRDS
    - access data set 251
- serial number volume label 107
- service routines, user supplied
  - using service vector 402
  - restrictions 402
- service vector, using 402
- shared
  - between tasks
    - data 425
    - files 425
    - VSAM data set 227
- shift code compilation 12
- simplification of expressions 310, 311
- SIR\_A\_DATA (parameter of IBMBSIR) 268
- SIR\_ENTRY (parameter of IBMBSIR) 268
- SIR\_FNCCODE (parameter of IBMBSIR) 267
- SIR\_MOD\_DATA (parameter of IBMBSIR) 268
- SIR\_RETCODE (parameter of IBMBSIR) 268
- SIS option 111, 222

- SIZE
  - compile-time option 22
  - MVS compilation 69
- SKIP
  - MVS, using for terminal input 92
  - VM, using for terminal input 88
- SKIP option 222
  - ENVIRONMENT option 111
  - in stream I/O 131
  - skip 32
- SKIPREC sort option 353
- SMESSAGE option 23
- sort program
  - assessing results 358
  - calling 355
  - CHKPT option 353
  - choosing type of sort 349
  - CKPT option 353
  - data input and output 360
  - description of 348
  - DYNALLOC option 353
  - E15 input handling routine 361
  - EQUALS option 353
  - FILSZ option 353
  - maximum record length 354
  - PLISRT 348
  - PLISRTA(x) command, examples 365—370
  - preparation 348
  - RECORD statement 361
  - RETURN statement 361
  - SKIPREC option 353
  - SORTCKPT 360
  - SORTCNTL 360
  - SORTIN 359
  - sorting field 352
  - SORTLIB 359
  - SORTOUT 359
  - SORTWK 355, 359
  - storage
    - auxiliary 355
    - main 355
  - write input/output routines 360
- sorting data 348
- source code, incorporating into program 62
- source key
  - in REGIONAL(1) data sets 190
  - in REGIONAL(2) data sets 195
  - in REGIONAL(3) data sets 202
  - indexed data sets 163
- source listing
  - location 16
  - statement numbers 18
- SOURCE option 23
- source program
  - compiler list 34
  - data set 65
  - source program (*continued*)
    - identifiers 8
    - included in compiler list 13
    - list 23
    - position within record 28
    - preprocessor 15
    - to shift outside text 16
  - source statement library 67
  - SPACE parameter
    - for stream I/O 133
    - library 124
    - standard data sets 64
    - stream I/O 136
  - spanned records 105
  - special case code for DO statement 313
  - spill file 66
  - SPROG suboption of LANGLVL 14
  - STACK run-time option 306
  - STACK subparameter 108
    - consecutive data set 157
  - standard data set 64
  - statement
    - lengths 66
    - nesting level 36
    - numbers
      - run-time messages 12
    - offset addresses 40
  - statements, transferring constant outside of loops 312
  - static
    - internal control section length 42
    - internal control section list 44
    - internal storage map 43
    - internal variable location 15
    - storage
      - list 15
      - show organization 15
- STATIC attribute, restriction on INITIAL attribute 333
- static CSECT, size of 331
- static information
  - on compiled modules 265, 270
    - compiled with TEST option 266
    - IBMBSIR 265
    - obtaining 265, 266
    - specifying type of 267
    - using IBMBSIR 266
  - on hooks 265
    - IBMBHIR 265
    - in modules compiled with TEST option 271
    - obtaining 265
  - retrieval module 266, 270
- step abend 108
- STMT
  - compile-time option 23
  - hooks location suboption 26
- storage
  - ABEND80A during compilation 69

- storage (*continued*)
  - blocking print files 139
  - control 332
  - DBCSOS ordering product 22
  - indexed data sets 163, 173
  - insufficient 22
  - library data sets 124
  - list object module storage requirement 23
  - requirements for compiler list 39
  - sort program 355
    - auxiliary storage 355
    - main storage 355
  - standard data sets 64
  - static storage
    - list 15
    - show organization 15
  - STORAGE option 23
  - to reduce requirement 20
  - TRANSIENT files 107
- storage allocation for automatic variables 331
- storage classes
  - AUTOMATIC
    - initialization of array elements 313
    - initialization of structure elements 313
  - BASED
    - initialization of array elements 313
    - initialization of structure elements 313
  - CONTROLLED
    - initialization of array elements 313
    - initialization of structure elements 313
  - for based and controlled variables on a virtual storage system 307
  - INITIAL attribute
    - initialization of array elements 313
    - initialization of structure elements 313
- stream and record files 144, 146
- STREAM attribute 129
- stream I/O 114, 129
  - data set
    - access 136
    - create 132
    - record format 137
  - DD statement 134, 138
  - ENVIRONMENT options 130
  - file
    - define 130
    - PRINT file 138
    - SYSIN and SYSPRINT files 142
  - performance improvement 341
- STREAM-oriented data transmission, maximizing input/output statements 342
- string
  - overlay defining 342
- string built-in functions, conditions for handling in-line 339
- string descriptors
  - bit offset component 383
  - concatenation with array descriptor 382
- string expressions, lengths of the intermediate results 327
- string handling, in-line operations for 328
- string locator/descriptors
  - allocated length component 383
  - bit offset 383
  - contents of 383
  - format of 383
  - in structure descriptors 384
  - string address 383
  - string length 383
  - varying marker 383
- string manipulation, in-line code for 314
- strings
  - addresses 383
  - bit offset 383
  - graphic string constant compilation 12
  - length 383
  - locator/descriptors 383
  - STRINGRANGE condition, prefix 305
  - varying marker 383
- strings, efficiency of 325
- structure and array assignments 313
- structure descriptors
  - contents of 384
  - descriptor components 384
  - example of 385
  - format of 384, 385
  - offset components 384
- structures
  - assignments, optimization of 313
  - base element offsets 384
  - base elements of, descriptors of 384
  - declaring, for bit-string data 324
  - descriptors 384
  - element assignments 313
  - initialization, optimization of 313
  - matching 323
  - names in data list 342
  - of arrays
    - aggregate locators for 385
    - structure descriptors for 385
  - optimization of
    - element assignments 313
    - initialization 313
- SUB control character 103
- SUBMIT command, description of 64
- subscript compatibility 8
- SUBSCRIPTRANGE condition, prefix 305
- subscripts
  - optimization of constants in expressions 311
  - uninitialized, detecting 323
  - using common expressions 310

- SUBSTR pseudovariable, varying string
  - assignments 339
- success in sorting 358
- SYMBOL ESD heading 41
- symbol table 26
- symbolic parameter in cataloged procedure 53
- synchronization for multitasking 424
- SYNTAX option 24
- syntax, diagrams, how to read xviii
- SYS1.PROCLIB (system procedure library) 123
- SYS1.SIBMTASK (multitasking) 53
- SYSCHK default 439, 440
- SYSCIN 65
- SYSIN 65
- SYSIN and SYSPRINT files 142
- SYSLIB
  - %INCLUDE 31
  - multitasking programs 53
  - preprocessing 67
- SYSLIN 65
- SYSOUT 132, 359
- SYSPRINT
  - compiler command 61
  - run-time considerations
- SYSPUNCH 65
- system
  - failure 440
  - restart after failure 440
  - SYSTEM compile-time options
    - NOEXECOPS 24
    - SYSTEM(CICS) 24
    - SYSTEM(CMS) 24
    - SYSTEM(CMSTPL) 24
    - SYSTEM(IMS) 24
    - SYSTEM(MVS) 24
    - SYSTEM(TSO) 24
    - type of parameter list 24
  - SYSTEM option 419
- SYSUT1
  - compiler data set 66

**T**

- tab control table 140
- TASK option 423
- tasking facilities 422
- TCAM message processing program 260, 261
- teleprocessing data sets
  - condition handling 260
  - defining files for 257
  - message control program (MCP) 255
  - specifying ENVIRONMENT options 257
- TCAM message processing program
  - discussion of 256
  - example of 261
  - writing 258
- teleprocessing data sets (*continued*)
  - teleprocessing organization 256
  - TRANSIENT file attribute 256
- temporary workfile
  - statement length 66
  - SYSUT1 66
- TERM extended parameter list request 390
- terminal
  - input 143
    - capital and lowercase Letters 145
    - COPY option of GET statement 145
    - end of file 145
    - format of data 143
    - stream and record files 144
    - using files conversationally 143
  - output 145
    - capital and lowercase characters 146
    - format of PRINT files 145
    - interactive program 146
    - output from PUT EDIT command 146
    - stream and record files 146
- TERMINAL option 25
- termination
  - compilation 10
  - tasks 426
- test
  - preprocessor 31
- TEST option
  - compile-time option 26
  - use in static information retrieval 266, 271
- testing, program, preprocessor 31
- TEXT file name 17
- TIME parameter 54
- TITLE option 93, 100, 257, 340
- TOTAL option 111, 151
- TP option 257
- tracing flow of control 272
- track index 166
- trailer label 107
- transfer
  - invariant expressions 321
- transfer of expressions from loops 312
- transferring constant expressions or statements outside of loops 312
- transferring expressions out of loops, undesired effects 312
- TRANSIENT file 256, 259
- TRKOFL option 111, 113, 120
- TRTCH subparameter 108
  - consecutive data set 157
- TSO
  - compiling 56
    - allocating data sets 57
    - background region 63
    - compiler listing 62
    - PLI command 59

- TSO (*continued*)
    - minimum region size
  - tuning a PL/I program
    - decreasing storage requirements
      - avoiding GOSTMT and GONUMBER 306
      - removing debugging aids 305
      - removing PUT DATA statements 306
    - items to remove
      - debugging aids 305
      - PUT DATA statements 306
    - looking for alternative source code 306
    - specifying run-time options 306
    - using in-line operations 306
    - using RPTSTG run-time option 306
  - tuning a program 305
  - tuning a program for a virtual storage system
    - assigning storage classes for based and controlled variables 307
    - avoiding large branches in source code 307
    - controlling the positioning for variables 308
    - designing and programming modular programs 307
    - handling aggregates larger than page size 307
    - making static internal CSECT read-only 308
    - minimizing paging 307
    - placing CSECTs within pages 308
    - placing variables within CSECTs 308
  - tuning run-time behavior 265—304
  - TXTLIB
    - compiling program for
      - description of 77
      - required commands 77
  - TYPE
    - ESD heading 41
    - PLIOPT command option 75
    - RECORD option for sort 354
  - type of sort, choosing 349
- U**
- U compiler message 44
  - U option of ENVIRONMENT
    - for record I/O 113
    - for stream I/O 130
  - UNALIGNED attribute 325
  - unaligned data fields 325
  - unblocked records for indexed data set 177
  - undefined-length records 106
  - UNDEFINEDFILE condition
    - BLKSIZE error 116
    - DD statement error 100
    - line size conflict in OPEN 139
    - OPEN error 151
  - UNIT parameter
    - accessing data set 132
    - consecutive data sets 159
    - for consecutive data sets 156
  - UNIT parameter (*continued*)
    - stream I/O 133, 136
  - unreferenced identifiers 8
  - UNSPEC pseudovisible, expression conversion 339
  - update
    - ESDS 231
      - indexed data set 179
        - direct access 181
        - sequential access 180
      - REGIONAL(1) data set 194
      - REGIONAL(2) data set
        - directly 200
        - sequentially 201
      - REGIONAL(3) data set
        - directly 206
        - sequentially 207
      - relative-record data set 252
    - use of additional variables to avoid conversions 326
    - user exit
      - preinitialized program 419
      - sort 351
    - using cataloged procedures, MVS 46
    - using during compilation 64
- V**
- V option of ENVIRONMENT
    - for record I/O 113
    - for stream I/O 130
  - V picture specification 343
  - variable-length records 104, 105
    - ASCII records 106
    - sort program 369
    - spanned records 105
  - variables
    - aliased, inhibiting optimization 318
    - arithmetic, declaring in nested blocks 316
    - automatic, storage allocation for 331
    - based, inhibiting optimization 318
    - optimization of
      - global 316
      - local 316
    - precision of 324
    - scale factor of 324
    - unpredictable values 323
  - VB option of ENVIRONMENT
    - for record I/O 113
    - for stream I/O 130
  - VB-format records 104
  - VBS option of ENVIRONMENT
    - for record I/O 113
  - VBS-format records 104
  - virtual storage system 307
    - tuning a program for 307, 308
      - assigning storage classes for based and controlled variables 307
      - avoiding large branches in source code 307

- virtual storage system (*continued*)
  - tuning a program for (*continued*)
    - controlling the positioning for variables 308
    - designing and programming modular programs 307
    - handling aggregates larger than page size 307
    - making static internal CSECT read-only 308
    - minimizing paging 307
    - placing CSECTs within pages 308
    - placing variables within CSECTs 308

## VM

- calling preinitialized program under VM 419
- commands 83
- compiling 73
- files, example of 82
- minimum region size
- PLIOPT command 73
- preinitialized program 419
- run-time considerations
  - automatic prompting 86
  - changing PRINT file format 86
  - description of 80
  - formatting conventions 86
  - input restrictions 86
  - OS data sets 84
  - output restrictions 86
  - PL/I conventions 86
  - record I/O at terminal 85
  - restrictions 85
  - stream I/O conventions 86
  - using data sets and files 81
- VSAM data sets, example of 83

## VMARGINS

- input record option 28

## VOLUME parameter

- consecutive data sets 133, 159
- for consecutive data sets 156
- for creating a data set 132
- for stream I/O 133
- stream I/O 136

## volume serial number 107

- consecutive data sets 133, 157
- indexed data sets 173
- regional data sets 208

## VS option of ENVIRONMENT

- for record I/O 113

## VS-format records 104

## VSAM (virtual storage access method)

- adapt existing program
  - CONSECUTIVE file 226
  - INDEXED file 226
  - REGIONAL(1) file 226
- alternate index path with a file 211
- data sets
  - alternate index paths 223
  - blocking 212
  - choosing a type 216

## VSAM (virtual storage access method) (*continued*)

### data sets (*continued*)

- defining 227
- defining files for 218
- dummy data set 217
- entry-sequenced 228
- file attribute 218
- key-sequenced and indexed
  - entry-sequenced 232
- keys for 215
- organization 212
- performance options 223
- relative record 247
- running a program with 211
- specifying ENVIRONMENT options 219
- types of 214
- using 211
- using with VM 83

### defining files 218

- ENV option 219
- for alternate index paths 223
- performance option 223

### files defined for non-VSAM data set 224

- adapting existing programs 225
- compatibility interface 225
- CONSECUTIVE files 224
- INDEXED files 224
- several files in one VSAM data set 226
- shared data sets 227

### indexed data set

- load statement and options 232
- mass sequential insert 238
- relative-record data set 249
- VSAM option 223

## VSEQUENCE 21

## VTOC 107

## W

- W compiler message 44
- WAIT statement 424
- warning, about IBMBHKS 266
- weak external reference ESD entry 42
- WHILE option of DO statement 321
- work data sets for sort 359

## X

- XCTL macro instruction 32
- XREF option 27
- XTENT option 187

## Z

- zero value 115—118