SC33-0025-2
File No. S360/S370-29

Program Product

OS
PL/I Optimizing Compiler: Execution Logic

Program Numbers 5734-PL1
5734-LM4
5734-LM5

(These program products are available as composite package 5734-PL3)

IBM

Third Edition (April 1973)

This is a major revision of and obsoletes SC33-0025-0 and SC33-0025-1. Information has been included on the new features that are available with release 2 of the PL/I Optimizing Compiler as follows:

COUNT option	Chapter 7
VSAM data sets	Chapter 8
ASSEMBLER option	Chapter 13

A number of minor changes and corrections have also been made throughout the book. A new topic heading "How Addressed" has been added to the control block descriptions in appendix A. Technical changes are marked with a vertical line to the left of the change.

This edition applies to Version 1 Release 2 Modification 0 of the OS Optimizing Compiler and to all subsequent releases until otherwise indicated in new editions or Technical Newsletters.

Changes will continually be made to the information herein; before using this publication in connection with the operation of IBM systems, consult the latest IBM System/360 and System/370 Bibliography Order No. GA22-6822, and associated Technical Newsletters for the editions that are applicable and current.

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM United Kingdom Laboratories Ltd., Programming Publications, Hursley Park, Winchester, Hampshire, England. Comments become the property of IBM.

IBM is grateful to the American National Standards Institute (ANSI) for permission to reprint its definitions from the American National Standard Vocabulary for Information Processing (Copyright 1970 by American National Standards Institute, Incorporated), which was prepared by Subcommittee X3.5 on Terminology and Glossary of American National Standards Committee X3.

© Copyright International Business Machines Corporation 1971,1972

Preface

The main purpose of this publication is to explain, in general terms, the way in which programs compiled by the OS PL/I Optimizing Compiler (Program Number 5734-PL1) are executed. It describes the organization of object programs produced by the compiler, the contents of the load module, and the main storage situation throughout execution. The information provided is intended primarily for those involved in maintenance of the compiler and its related library program products. The publication will also provide valuable information for applications programmers, since a knowledge of the way in which source program statements are executed will lead to the writing of more efficient programs. The book also contains a chapter on how to obtain and read a PL/I dump.

Although different source programs produce different executable programs, the structure of every executable program produced by the compiler is basically the same. This structure is explained in chapter 1. Chapters 2,3,4, and 5 describe the various elements that make up the load module. Chapters 6 and 7 explain the housekeeping and error-handling schemes. Chapters 8, 9, 10, and 11 describe the implementation of various language features, the majority of which are handled by a combination of compiled code, PL/I library routines, and Operating System routines. Chapter 12 is the guide to obtaining and using dumps. Chapter 13 deals with interlanguage communication. The final chapter, chapter 14, discusses those aspects of execution that apply only to a multitasking environment. In addition, appendix A contains details of all control blocks that can exist during execution.

The reader of this publication is assumed to have a sound knowledge of PL/I, and a working knowledge of the IBM System/360 Operating System and its assembler language. It is recommended, therefore, that the reader should be familiar with the content of the following publications:

RECOMMENDED PUBLICATIONS

OS PL/I Checkout and Optimizing Compilers: Language Reference Manual, Order No. SC33-0009

System/360 Principles of Operation, Order No. GA22-6821

Introduction to System Control Programs, Order No. GY24-5017

REFERENCE PUBLICATIONS

This book makes reference to the following publications for related information that is beyond its scope:

OS PL/I Optimizing Compiler:

Programmer's Guide, Order No. SC33-0006

Program Logic, Order No. LY33-6007

System Information, Order No. SC33-0026

Supervisor and Data Management Macro Instructions, Order No. GC28-6647

IBM System/360 Reference Data Card, Order No. GX20-1703

OS PL/I Resident Library: Program Logic, Order No. LY33-6008

OS PL/I Transient Library: Program Logic, Order No. LY33-6009

OS Programmer's Guide to Debugging, Form C28-6670

OS Linkage Editor and Loader, Order No. GC28-6538

OS Supervisor and Data Management Macro Instructions, Order No. GC28-6647

AVAILABILITY OF PUBLICATIONS

The availability of a publication is indicated by its use key, the first letter in the order number. The use keys are:

G - General: available to users of IBM systems, products, and services without charge, in quantities to meet their normal requirements; can also be purchased by anyone through IBM branch offices.

L - Licensed materials, property of IBM: available only to licensees of the related program products under the terms of the license agreement.

S - Sell: can be purchased by anyone through IBM branch offices.

Contents

CHAPTER 1: INTRODUCTION 1
Processing a PL/I Program 1
Compilation 1
Link-editing 1
Execution 1
Factors Affecting Implementation . . 3
Key Features of the Executable Program 3
Communications Area 3
Dynamic Storage Allocation 3
Use of Library Subroutines 5
Initialization/Termination Routines 5
Contents of a Typical Load Module . . 5
The Overall use of Storage 8
The Process of Execution 8

CHAPTER 2: COMPILER OUTPUT 11
Introduction 11
The Organization of this Chapter 12
Listing Conventions 12
Static-Storage Map 15
Object-Program Listing 15
Static Internal Control Section . . . 18
Program Control Section 18
Register Usage 18
Dedicated Registers 19
Work Registers 19
Floating-Point Registers 19
Library Register Usage 19
Handling and Addressing Variables and Temporaries 19
Automatic Variables 19
Compiler-generated Temporaries . . 20
Temporaries for Adjustable Variables 20
Controlled Variables 20
Based Variables 20
Static Variables 21
Addressing Beyond the 4K Limit . . 21
The Pseudo-Register Vector (PRV) . 21
Addressing Controlled Variables and Files 21
The Location of the PRV 22
Initialization of the PRV . . . 22
Program Control Data 22
Handling Data Aggregates 23
Arrays of Structures and Structures of Arrays 23
Array and Structure Assignments . 23
Handling Flow of Control 24
Activating and Terminating Blocks 24
Prologue and Epilogue Code 26
Prologue 26
Epilogue 26
CALL Statements 27
Function References 27
END Statement 27
RETURN Statement 28
GOTO Statements 28
GOTO within a Block 28
GOTO out of Block 29
GOTO Label Variable 29
Errors when Using Label Variables 30
GOTO-only On-Units 30
Interpretive GOTO routines . . . 30
Argument and Parameter Lists 30
Library Calls 31
Setting-Up Argument Lists . . . 31
Addressing the Subroutine . . . 32
DO-loops 32
Compiler-generated Subroutines . . . 33
Optimization and its Effects 33
Examples of Optimized Code 33
Elimination of Common Expressions 34
Movement of Expressions out of Loops 34
Elimination of Unreachable Statements 36
Simplification of Expressions . 37
Modification of DO-loop Control Variables 37
Branching around Redundant Expressions 37
Rationalization of Program Branches 38
Use of Common Constants and Control Blocks 38

CHAPTER 3: THE PL/I LIBRARIES . . . 41
Resident and Transient Libraries . . 41
Naming Conventions 41
The Multitasking Library 42
Library Workspace 42
Format of Library Workspace . . . 42
Allocation of Library Workspace . 44
Library Modules and Weak External References 44
The Shared Library 44
Communication between Program Region and Link-Pack-Area . . . 48
Execution when Using the Shared Library 48
Program Initialization 48
Initializing the Shared Library 50
Multitasking Considerations . . 50

CHAPTER 4: COMMUNICATION BETWEEN ROUTINES 53
Notes on Terminology 53
Descriptors and Locators 53
String Locator/Descriptor . . . 55
Area Locator/Descriptor 55
Aggregate Locator 55
Array Descriptor 55
Structure Descriptor 55
Aggregate Descriptor Descriptor 57
Arrays of Structures and Structures of Arrays 57
Data Element Descriptors 57

Symbol Tables and Symbol Table Vectors 61

CHAPTER 5: OBJECT PROGRAM INITIALIZATION 65
Link-editing 65
Program Initialization 65
Initialization and Termination Routines 66
Resident Initialization/Termination Routine IBMBPIR 66
The Process of Initialization . 68
Handling Execution Time Options 68
Acquiring the ISA 68
Initialization of the Program Management Area 68
Initializing PL/I Error Handling 68
Error Situations 69
The Process of Termination . . . 69
The Program Management Area . . . 69
Task Communications Area (TCA) . 69
TCA Implementation Appendage . . 70
Save Area for IBMBPGR 71
Dummy ONCA 71
Translate-and-Test Table 71
Dump File Block 71
Loaded Module or Ordered Delete List 71
Dummy Tasks and Event Variables 71
Diagnostic File Block 72
Dummy DSA 72
Library Workspace (LWS) 72
ON Communications Area (ONCA) . 72
Pseudo-Register Vector 72
Multitasking 72

CHAPTER 6: STORAGE MANAGEMENT . . . 75
Types of Dynamic Storage Required 75
Contents of LIFO (Last-In/First Out) Storage 75
Contents of Non-LIFO Storage . . 75
Dynamic Storage Allocation 76
Fields Used in Storage Handling 76
Allocating and Freeing LIFO Storage 76
Allocating and Freeing Non-LIFO Storage 79
Acquiring a New Segment of LIFO Storage 79
IBMBPGR - Storage Management Routine 79
Allocating Non-LIFO Storage (IBMBPGRA) 80
Freeing Non-LIFO Storage (IBMBPGRB) 80
Segment Handling (IBMBPGRC and IBMBPGRD) 80
Storage Reports 82
Action during Initialization . . 83
Action during Execution 83
Action on Termination 83
Storage Reports for Multitasking Programs 84
Storage Management in Programmer-allocated Areas 84
Multitasking Considerations 85
Acquiring the ISA when Multitasking 85

CHAPTER 7: ERROR AND CONDITION HANDLING 87
Terminology 87
Background to Error Handling 88
System Facilities 88
PL/I Facilities 88
Implementation of Error Handling . . 93
Detecting the Occurrence of Conditions 96
System Detected Conditions 96
Software Detected Conditions . . . 96
Detecting I/O Conditions 96
Executing Signal Statements . . . 96
Passing Information about Interrupt 96
Error Code 97
Condition Built-in Functions . . 97
Chain of CNCAs 97
Establishment and Enablement Information 99
Enablement 99
Qualified Conditions 100
Establishment - Executing ON and REVERT Statements 100
Qualified Conditions 100
Unqualified Conditions 102
Handling On-units 102
The Logic of the Error Handler . . . 102
IBMBERR - Error-handling Module . 102
Program Check Interrupts 103
Software Interrupts 103
Return to Point of Interrupt . . . 104
Software Interrupts 104
Program Check Interrupts 105
The Check Condition 105
Raising the Check Condition . . 105
Testing for Enablement 106
Searching for Established On Units 106
Standard System Action 106
Error Messages 106
Message Formats 106
Interrupts in Library Modules . 107
Identifying the Erroneous Statement 107
Identifying Entry Point Name and Statement Number 107
Filename and Name of CONDITION Condition 108
Message Text Modules 108
Diagnostic File Block 108
Dump Routines 109
Dump File 109
Miscellaneous Error Modules 112
Abend Analyzers 112
Exceptional Error Message Modules 112
The FLOW and COUNT Options 114
Implementation of FLOW and COUNT . 115
Tables Used by FLOW and COUNT . 115
Executable Code for FLOW and COUNT 115
Action During Compilation . . . 117
Action During Program Initialization 118
Action During Execution 118
Action on Output 120

CHAPTER 8: RECORD-ORIENTED INPUT/OUTPUT 125
Introduction 125
Summary of Record I/O Implementation 125
File Declarations 125
OPEN Statements 125
Transmission Statements 125
CLOSE Statements 127
Implicit Open 127
Implicit Close 127
Access Method 127
File Declaration Statements 127
Execution 127
OPEN Statement 130
Execution 130
Actions Carried Out by Transient Open Routines 130
VSAM Data Sets 130
The FCB and File Addressing . . 133
Transmission Statements (Library-Call I/O) 135
Compiler Output 135
Execution 138
Transmitter Action 138
EVENT Option 138
Execution 140
Use of the IOCB 140
Allocation of IOCBs 140
IOCBs and Dummy Records 140
Raising Conditions in Event I/O 140
Exclusive I/O 140
CLOSE Statements and Implicit Close . 143
Compiler Output 143
Execution 143
Implicit Open for Library-Call I/O . 143
Compiler Output 143
Execution 143
Error Conditions in Transmission Statements 144
General Error Routines (Transient) 146
ENDFILE Routine 146
TRANSMIT Condition 146
In-line I/O Statements 146
Control Blocks 146
Executable Instructions 146
Error Conditions 146
Implicit Open for In-Line Calls 147

CHAPTER 9: STREAM-ORIENTED INPUT/OUTPUT 153
Note on Terminology 153
Introduction 153
Operations in a Stream I/O Statement 153
Stream I/O Control Block (SIOCB) 154
File Handling 154
Transmission 154
Opening the File 156
Implicit Open 156
Keeping Track of Buffer Position 156
Enqueuing and Dequeuing on SYSPRINT 156
Handling the Conversions 160
Handling GET and PUT Statements . . . 160
List-directed GET and PUT Statements 160
PUT LIST Statement 160
GET LIST Statement 168
Data-directed GET and PUT Statements 168
Identifying the Name 169
Edit-directed GET and PUT Statements 169
Compiler-generated Subroutines . 169
Handling Control Format Items . 170
Matching and Non-Matching Data and Format Lists 170
Formatting for Print Files 170
Handling Format Options 171
Input and Output of Complete Arrays . 174
PL/I Conditions in Stream I/O 174
TRANSMIT Condition 174
CONVERSION Condition 174
NAME Condition 174
ENDFILE Condition and Unexpected End of File 174
Built-in Functions in Stream I/O . . 174
The COPY Option 175
Handling the Copy File 175
The STRING Option 175
Completing String-handling Operations 177
The Time-Sharing Option (TSO) and Conversational Files 177
Conversational Transmitter Modules 177
Output Transmitter IBMBSOC . . . 177
Input Transmitter IBMBSIC . . . 177
Formatting 178
Formatting Module IBMBSPC . . . 178
Summary of Subroutines Used 178
Initializing Modules 178
Director Modules 179
Library Director Routines . . . 179
Modules Used with Compiler generated Subroutines 179
Module for Complete Library Control of Edit-directed I/O of a Single Item 179
Compiler-generated Director Routines 179
Transmitter Modules 179
Formatting Modules 180
Library Subroutines 180
Compiler-generated Subroutine . 180
External Conversion Director Modules 180
Conversational Modules 180
Miscellaneous Modules 180

CHAPTER 10: DATA CONVERSION 181
Note on Terminology 181
The Library Conversion Package . . . 181
Conversion Module Naming Conventions 182
Specifying a Conversion Path . . . 182
Housekeeping when more than one Module is Used 182
Arguments Passed to the Conversion Routines 182
Communication between Modules . . 184
Free Decimal Format 184
In-Line Conversions 185
Note about Picture Variables . . 185
Example: Fixed-Binary to Fixed Decimal (Compiler Conversion No. 6) 186
Multiple Conversions 187

Hybrid Conversion 187
Raising the CONVERSION Condition . . 188

CHAPTER 11: MISCELLANEOUS LIBRARY SUBROUTINES AND SYSTEM INTERFACES . 189
Computational and Data-handling Subroutines 189
Arithmetic and Mathematical Subroutines 189
Array, String, and Structure Subroutines 189
Handling Interleaved Arrays (IBMBAIH) 190
Structure Mapping (IBMBAMM) . . 191
Miscellaneous System Interfaces . . . 191
Time 191
Date 192
Delay 192
Display 192
Sort/Merge 192
Housekeeping Problems 194
Restoration of the PL/I Environment on Exit from SORT . 194
Summary of Work Done by the SORT Module 194
Storage for SORT 197
Checkpoint/Restart 197
Wait 197
Event Variables 197
WAIT Statement (Non Multitasking) 198
Housekeeping Problems 198
Control Blocks 202
Wait Module (IBMBJWT) 202

CHAPTER 12: DEBUGGING USING DUMPS . 205
How to use this Chapter 205
Section 1: How to Obtain a PL/I Dump 207
Call PLIDUMP 207
Recommended Coding 207
Avoiding Re-compilation 208
Contents of a PL/I Dump 208
Headings 208
Trace Information 210
File Information 211
Hexadecimal Dump 212
Block Option 212
Section 2: Recommended Debugging Procedures 212
Debugging Procedures 215
PL/I Dump Called from On-Unit . 215
OS ABEND Dump 215
Section 3: Locating Specific Information 217
Contents 217
Key Areas of a PL/I Dump 217
Key Areas of an ABEND Dump . . . 217
Stand-alone Dumps 217
Housekeeping Information in all Dumps 217
Finding Variables 217
Control Blocks and Fields . . . 218
Key Areas of a PL/I Dump 218
P1: Statement Number and Address where Error Occurred (Dump Called from On-Unit only) . . . 218
P2: Type of Error (Applies to Dump Called from On-Unit only) 218
P3: Register Contents at Time of Error or Dump Invocation . . 218
P4: The DSA Chain 219
P5: The TCA 219
Key Areas of an ABEND Dump 219
O1: Address of Interrupt . . . 219
O2: Type of Interrupt 219
O3: Register Contents at the Point of Interrupt 219
O4: The DSA Chain 221
O5: The TCA 221
O6 Finding the Program Interrupt Element (PIE) 221
Stand-alone Dumps 221
S1: Finding Key Areas in Stand alone Dumps 221
Housekeeping Information in all Dumps 221
H1: Following the DSA Backchain 221
H2: Associating Instruction with Correct Statement and Program Block 221
H3: Following Calling Trace . . 223
H4: Associating DSA with Block 223
H5: Finding Relevant ONCA . . . 223
H6: Following the Chain of ONCAs 223
H7: Finding Information from IBMBERR's DSA 223
H8: Finding and Interpreting Register Save Areas 223
H9: Register Usage 224
H10: Following Free-Area Chain 224
H11: Finding the Task Variable 224
H12: Block Structure of Program (Static Backchain) 224
H13: Forward Chain in DSAs . . 224
H14: Action if Error is in a Library Module 224
H15: Discovering Contents of Parameter Lists 225
H16: Finding Main Procedure DSA 225
H17: Finding the Relationship between Tasks 225
To Find the Parent Task 225
To Find all Subtasks of a Task . 225
To Find Sister Tasks 225
H18: Finding the Tasking Appendage 225
H19: Finding the TCA from the Tasking Appendage 226
Finding Variables 226
V1: Automatic Variables 226
V2: Static Variables 226
V3: Controlled Variables . . . 226
V4: Based Variables 226
V5: Area Variables 226
V6: Variables in Areas 226
Control Blocks and Fields 226
C1: Quick Guide to Identifying Control Fields 227
Special Considerations for Multitasking 228

CHAPTER 13: INTERLANGUAGE COMMUNICATION 231
Summary of Interlanguage Facilities 231

Background to Interlanguage Communication 231
Differences in Data Aggregates . . 233
Use of Locators 233
Differences of Environment 233
The Principles of Interlanguage Communication 233
PL/I Calls COBOL or FORTRAN . . 234
FORTRAN or COBOL Calls PL/I . . 234
Retaining the Environment . . . 234
Handling Changes of Environment . . . 238
Interlanguage Housekeeping Routines and their Control Blocks 238
Handling FORTRAN and PL/I Initialization/Termination Routines 238
Handling the INTER Option . . . 242
STOP and STOP RUN Statements . . 242
Housekeeping Module Descriptions . . 242
COBOL when Called from PL/I (IBMBIEC) 242
Before Entry to COBOL Program . 242
On Return from COBOL Program (IBMBIECC) 242
Action on Interrupt in COBOL with INTER 246
Zerodivide On-Units 246
Handling STOP RUN statements . . 246
FORTRAN when Called from PL/I (IBMBIEF) 246
Before Entry to the FORTRAN Program 246
Action on Return from FORTRAN Program (IBMBIEFC and IBMBIEFD) 247
Action on Interrupt in FORTRAN . 247
Termination of Caller 248
STOP Statements 248
PL/I Called from COBOL or FORTRAN (IBMBIEP) 248
Before Entry to PL/I Program (IBMBIEPA) 248
Action after the PL/I Program is Completed 249
Interrupt Handling 249
Termination of PL/I Environment 249
STOP and STOP RUN Statements . . 249
Handling Data Aggregate Arguments . . 250
Arrays 250
Structures 250
Methods Used to Handle Data Aggregate Arguments 250
NOMAP, NOMAPIN, and NOMAPOUT Options 250
Calling Sequence 251
ASSEMBLER Option 251
COBOL Option in the Environment Attribute 251

CHAPTER 14: MULTITASKING 255
Introduction 255
The Concept of the Control Task 255
Communication between Tasks . . 256
Holding the Priority of the Task 256
Multitasking Housekeeping 256
The Multitasking Library 260
How the Control Task Operates 265
Attaching a Task 265
Failure of CALL...TASK Statements 266
Detatching a Task 266
Abnormal Termination of a Task . 266
The Get-Control and Free-Control Routines 266
Altering COMPLETION and PRIORITY Values 268
Executing the WAIT Statement 268
The Wait Module IBMTJWT 269
Enqueuing and Dequeuing on SYSPRINT . 270

APPENDIX A: CONTROL BLOCKS 271
Area Locator/Descriptor 272
Area Descriptor 272
Area Variable Control Block 273
Aggregate Descriptor Descriptor . . . 274
Structure Element 274
Base Element 274
Aggregate Locator 275
Array Descriptor 276
Arrays of Strings or Areas . . . 276
Controlled Variable Block 277
Data Element Descriptor (DED) 278
Format of DEDs 278
General Format 278
DED for STRING data 279
DED for FLOAT Data 279
DED for FIXED Data 279
DED for PICTURE STRING Data . . 279
DED for PICTURE DECIMAL Arithmetic Data 279
DED for PROGRAM CONTROL Data . . 280
FORMAT DEDs - FEDs 280
DED for F and E FORMAT Items (FED) 280
DED for PICTURE FORMAT Arithmetic Items (FED) 280
DED for PICTURE FORMAT character Items (FED) 281
DED for C FORMAT Items (FED) . . 281
DED for CONTROL FORMAT Items (FED) 281
DED for STRING FORMAT Items (FED) 281
Declare Control Block (DCLCB) 282
Diagnostic File Block (DFB) 283
Dynamic Storage Area (DSA) 284
Dump Block (DUB) 286
Entry Data Control Block 287
Environment Block (ENVB) 288
Event Table (EVTAB) 289
Event Variable Control Block 290
Exclusive Block IOCB (XBI) 291
Exclusive Block File (XBF) 292
File Control Block (FCB) 293
Common Section 293
Record I/O Section 295
Stream I/O Section 296
Flow Statement Table 297
Where Held 297
Interlanguage Root Control Block (IBMBILC1) 298
Interlanguage VDA 299
Interrupt Control Block (ICB) 300
Input/Output Control Block (IOCB) . . 301
Key Descriptor (KD) 304

Label Data Control Block 305
Label Variable and Label temporary 305
Label Constant 305
Library Workspace (LWS) 306
On Communications Area (ONCA) 307
Dummy ONCA 307
On Control Block (ONCB) 308
Static and Dynamic ONCBs 308
Open Control Block 309
Ordered Delete List (ODL) 310
PLIMAIN 311
Dummy PLIMAIN 311
Record Descriptor (RD) 312
Request Control Block (RCB) 313
Statement Frequency Count Table . . . 314
Statement Number Table 315
Storage Report Table 316
Stream I/O Control Block (SIOCB) . . 317
String Locator/Descriptor 318
Structure Descriptor 319
Symbol Table (SYMTAB) 320
Symbol Table Vector 322
Task Communications Area (TCA) . . . 323
TCA Appendage (TIA) 325
TCA Tasking Appendage (TTA) 326
Task Variable (TV) 327
Wait Information Table (WIT) 328
Zygo-Lingual Control List (ZCTL) . . 329
INDEX 331

Figures

Figure 1.1. The process of running a PL/I program xiv
Figure 1.2. Use of PL/I dynamic storage 2
Figure 1.3. Contents of a typical load module 4
Figure 1.4. Use of storage 6
Figure 1.5. Flow of control during execution 7
Figure 2.1. The output from the compiler 10
Figure 2.2. Contents of listing and associated compiler options 12
Figure 2.3. Example of static storage listing 13
Figure 2.4. Part of an object program listing (For source see Figure 2.3) 14
Figure 2.5. Register usage in compiled code 17
Figure 2.6. Library register usage . 20
Figure 2.7. Use of the pseudo register vector (PRV) 22
Figure 2.8. Typical prologue code . 24
Figure 2.9. Contents of typical compiled code DSA 25
Figure 2.10. Epilogue code 27
Figure 2.11. Examples of library calling sequences 31
Figure 2.12. Mnemonic letters in library module entry-point names . . 32
Figure 2.13. Offsets where addresses of library modules are held in the TCA 32
Figure 2.14. Modification of do-loop control variable 35
Figure 2.15. Branching around redundant expressions 36
Figure 2.16. Use of common constants 36
Figure 3.1. Library module naming conventions 40
Figure 3.2. Library workspace . . . 43
Figure 3.3. Example of use of WXTRNs 45
Figure 3.4. The shared library during execution 46
Figure 3.5. The format of shared library modules 47
Figure 3.6. Addressing a module in the shared library 49
Figure 4.1. Example of descriptor, locator, DED, and storage location of an array 52
Figure 4.2. Descriptors, locators, and symbol tables: when generated, where held 54
Figure 4.3. String locator/descriptor 56
Figure 4.4. Area locator/descriptor 56
Figure 4.5. Aggregate locator . . . 56
Figure 4.6. Array descriptor 56
Figure 4.7. Aggregate descriptor descriptor 58
Figure 4.8. Example of handling a structure containing an adjustable extent 59
Figure 4.9. Structure descriptor for arrays of structures and structures of arrays 60
Figure 4.10. Format of DEDs 62
Figure 4.11. Symbol tables and symbol table vectors 63
Figure 5.1. Flow of control during execution 64
Figure 5.2. Program management area 67
Figure 6.1. Use of storage in the ISA 74
Figure 6.2. Principles involved in allocating and freeing LIFO storage 77
Figure 6.3. Principles involved in allocating and freeing non-LIFO storage 78
Figure 6.4. Format of element on free area chain 80
Figure 6.5. Principles involved in allocating and freeing segments of PL/I dynamic storage 81
Figure 6.6. Format of second and subsequent segments of the LIFO stack 82
Figure 7.1. The principles of error handling 86
Figure 7.2. Machine interrupts associated with PL/I conditions . . 88
Figure 7.3. (Part 1 of 2). PL/I conditions 89
Figure 7.3. (Part 2 of 2). PL/I conditions 90
Figure 7.4. Static and dynamic descendency 91
Figure 7.5. The major fields used in error handling 92
Figure 7.6. An example of error handling 94
Figure 7.7. Addressing on-units . . 95
Figure 7.8. Accessing a built-in function value from the chain of ONCAs 98
Figure 7.9. Meaning of enablement bits 100
Figure 7.10. Simplified flowchart of IBMBERR 101
Figure 7.11. Handling the CHECK condition 110
Figure 7.12. Interrelationship of dump routines 111
Figure 7.13 How branch counts are used to calculate the number of times each statement is executed. . 113
Figure 7.14. The contents of the flow statement table and the statement frequency count table. . . 116
Figure 7.15. Outline of error handling 122

Figure 8.1. The principles used in record I/O implementation 124
Figure 8.2. Library subroutines used in record I/O 126
Figure 8.3. Access methods and file types 127
Figure 8.4. (Part 1 of 2). The fields used in implementing record I/O 128
Figure 8.4. (Part 2 of 2). The fields used in implementing record I/O 129
Figure 8.5. Information in the file declaration is held in the ENVB and the DCLCB until the file is opened . 131
Figure 8.6. Open statement 132
Figure 8.7. Addressing files via DCLCB and PRV 134
Figure 8.8. (Part 1 of 2). Handling a transmission statement 136
Figure 8.8. (Part 2 of 2). Handling a transmission statement 137
Figure 8.9. Handling the EVENT option 139
Figure 8.10. The execution of an explicit CLOSE statement 141
Figure 8.11. The addressing mechanism used during implicit open 142
Figure 8.12. Record I/O error modules 144
Figure 8.13. The fields used in record I/O error handling 145
Figure 8.14. In-line I/O transmission statement 148
Figure 8.15. Overview of record I/O 149
Figure 8.16. Conditions under which I/O statements are handled in-line . 151
Figure 9.1. The principles used in stream I/O 152
Figure 9.2. Record boundaries do not affect stream I/O 154
Figure 9.3. Simplified flow diagram of a stream I/O statement 155
Figure 9.4. Stream I/O control block (SIOCB) 156
Figure 9.5 The use of FREM and FCBA in recording buffer position 157
Figure 9.6. (Part 1 of 2). Flow of control through a PUT LIST statement 158
Figure 9.6. (Part 2 of 2). Flow of control through a PUT LIST statement 159
Figure 9.7. Code generated for typical list-directed I/O statement 161
Figure 9.8. (Part 1 of 2). Handling a GET DATA statement 162
Figure 9.8. (Part 2 of 2). Handling a GET DATA statement 163
Figure 9.9. Typical data-directed code 164
Figure 9.10. The use of the library in edit-directed I/O 165
Figure 9.11. (Part 1 of 2). Edit directed statement with matching data and format lists 166
Figure 9.11. (Part 2 of 2). Edit directed statement with matching data and format lists 167
Figure 9.12. Code generated for an edit-directed statement with matching data and format lists . . . 172
Figure 9.13. Code sequences used for matching and non-matching data and format lists 173
Figure 9.14. The current buffer pointer FCBA and FCPM, the copy pointer, keep track of the data to be copied 176
Figure 10.1. Internal forms of data types 181
Figure 10.2. (Part 1 of 2). Data conversions performed in-line . . . 183
Figure 10.2. (Part 2 of 2). Data conversions performed in-line . . . 184
Figure 10.3. Fundamental in-line conversions 185
Figure 10.4. Multiple conversions . 187
Figure 11.1. Arithmetic operations performed by library subroutines . . 190
Figure 11.2. Array, structure, and string subroutines 191
Figure 11.3. Indexing interleaved arrays 193
Figure 11.4. DSA chaining during the execution of SORT 195
Figure 11.5. Summary of action during use of a SORT exit 196
Figure 11.6. Example of WAIT implementation problems 198
Figure 11.7. (Part 1 of 2). Summary of the wait statement 200
Figure 11.7. (Part 2 of 2). Summary of the wait statement 201
Figure 12.1. How to use this chapter when debugging 204
Figure 12.2. Code for debugging . . 206
Figure 12.3. Suggested method of obtaining a dump when re-compilation is particularly undesirable. (See text before using this me 208
Figure 12.4. An example of PLIDUMP . 209
Figure 12.5. Abbreviations for condition names used in PLIDUMP trace information. 210
Figure 12.6. Error code field lookup table 211
Figure 12.7. The contents of IBMBERR's DSA after a system detected and a PL/I interrupt . . . 213
Figure 12.8. The chaining of DSAs . 214
Figure 12.9. The register save area in the DSA 220
Figure 12.10. Register usage 224
Figure 13.1. The principles of interlanguage communication 230
Figure 13.2. Calling sequence when PL/I calls COBOL or FORTRAN 232
Figure 13.3. Code generated when PL/I calls a COBOL routine 235
Figure 13.4. The sequence of events when FORTRAN or COBOL calls PL/I . . 236
Figure 13.5. Chaining of save areas when PL/I is called from a COBOL or FORTRAN principal procedure 237
Figure 13.6. Example of chaining sequences (PL/I principal procedure) 239

Figure 13.7. Examples of chaining sequences (FORTRAN principal procedure) 241
Figure 13.8. The concept of save area rechaining (see figures 13.9 and 13.10 for details) 243
Figure 13.9. Rechaining of save areas when FORTRAN is called from PL/I and the FORTRAN environment needs initializing 244
Figure 13.10. Rechaining of save areas when PL/I is called from FORTRAN or COBOL and the environment requires initialization 245
Figure 14.1. Multitasking is implemented by use of a multitasking library 254
Figure 14.2. The hierarchy of tasks. 257
Figure 14.3. The functions of the control task 258
Figure 14.4. The post and wait ECBs 259
Figure 14.5. Modules in the multitasking library 260
Figure 14.6. Backchains in multitasking 261
Figure 14.7. The chaining of tasks through their tasking appendages . . 262
Figure 14.8. A simplified flowchart of IBMTPIR 263
Figure 14.9. Chains and pointers used in implementing the WAIT statement 267
Figure 14.10. Reusing event variables, and the need fpr the EVTAB chain 270

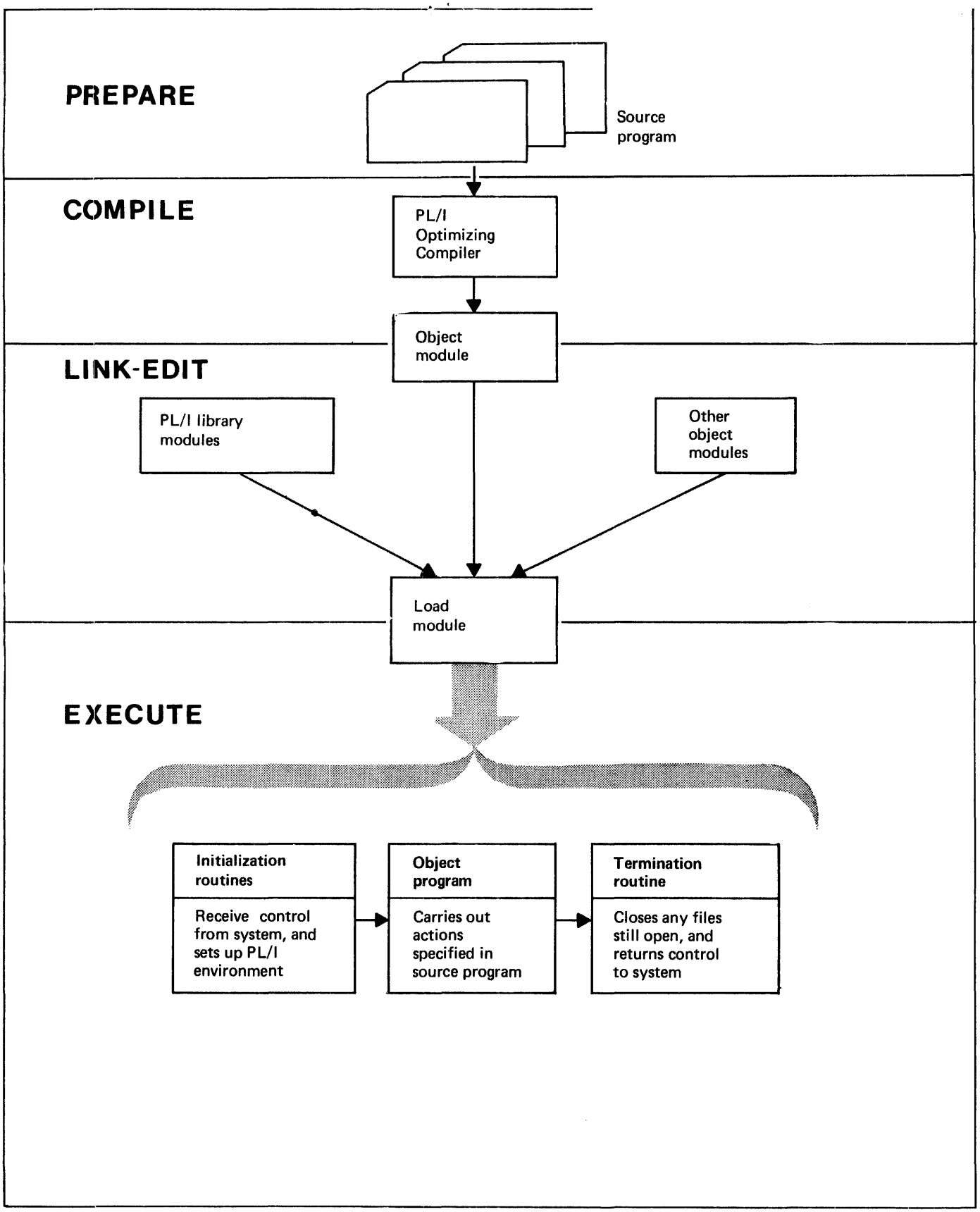


Figure 1.1. The process of running a PL/I program

Chapter 1: Introduction

Processing a PL/I Program

Figure 1.1 shows the processes through which a PL/I program passes from its inception to its use. There are four stages:

1. Writing the program and preparing it for the computer.
2. Compilation: translating the program into machine instructions (i.e., creating an object module).
3. Link-editing: producing a load module from the object module. This includes linking the compiled code with PL/I library modules, and possibly with other compiled programs. It also includes resolving addresses within the code.
4. Execution: running the load module.

The process is not necessarily continuous. The program may, for example, be kept in a compiled or link-edited form before it is executed, and it will normally be executed a number of times once compiled.

COMPILATION

Compilation is the process of translating a PL/I program into machine instructions. This is done by associating PL/I variables with addresses in storage and translating executable PL/I statements into a series of machine instructions. For example, the PL/I statements:

```
DCL I,J,K;
I=J+K;
```

would typically result in the generation of machine instructions corresponding to the assembler language instructions shown below:

```
LH    7,88(0,13)   Load J into register 7
AH    7,90(0,13)   Add K to J
STH   7,96(0,13)   Place result in I
```

(The variables I, J, and K are held at offsets 96,88, and 90, respectively, from the address in register 13.)

The OS PL/I Optimizing Compiler does not translate all PL/I statements directly into the necessary machine instructions. Instead, certain statements are translated into calls to standard subroutines held in the OS PL/I Resident Library. Some of the resident library routines may, in turn, call further library routines from either the resident or the transient PL/I library. The following PL/I statements would, for example, result in a call being made to a resident library routine.

```
DCL X,Y;
X=SIN(Y);
```

The code that would typically result from such statements is shown below:

```
LA    14,92(0,13)    Place address of Y
                     in register 14
LA    15,96(0,13)    Place address of X
                     in register 15
STM   14,15,80(0,3)  Place addresses in
                     argument list
LA    1,80(0,3)      Point register 1 at
                     argument list
L     15,88(0,3)     Load register 15
                     with the address of
                     the resident library
                     routine IBMBMGS.
                     (This is held in the
                     form of an address
                     constant generated
                     by the compiler and
                     resolved by the
                     linkage editor.)
BALR  14,15          Branch to the
                     library routine,
                     which will carry out
                     the required
                     function.
```

LINK-EDITING

Link-editing links the compiler output with external modules that have been requested by the compiled program. These will be PL/I resident library routines, and, possibly, modules produced by further compilations. As well as linking the external modules, the linkage editor also resolves addresses within the object module.

EXECUTION

The optimizing compiler produces code that

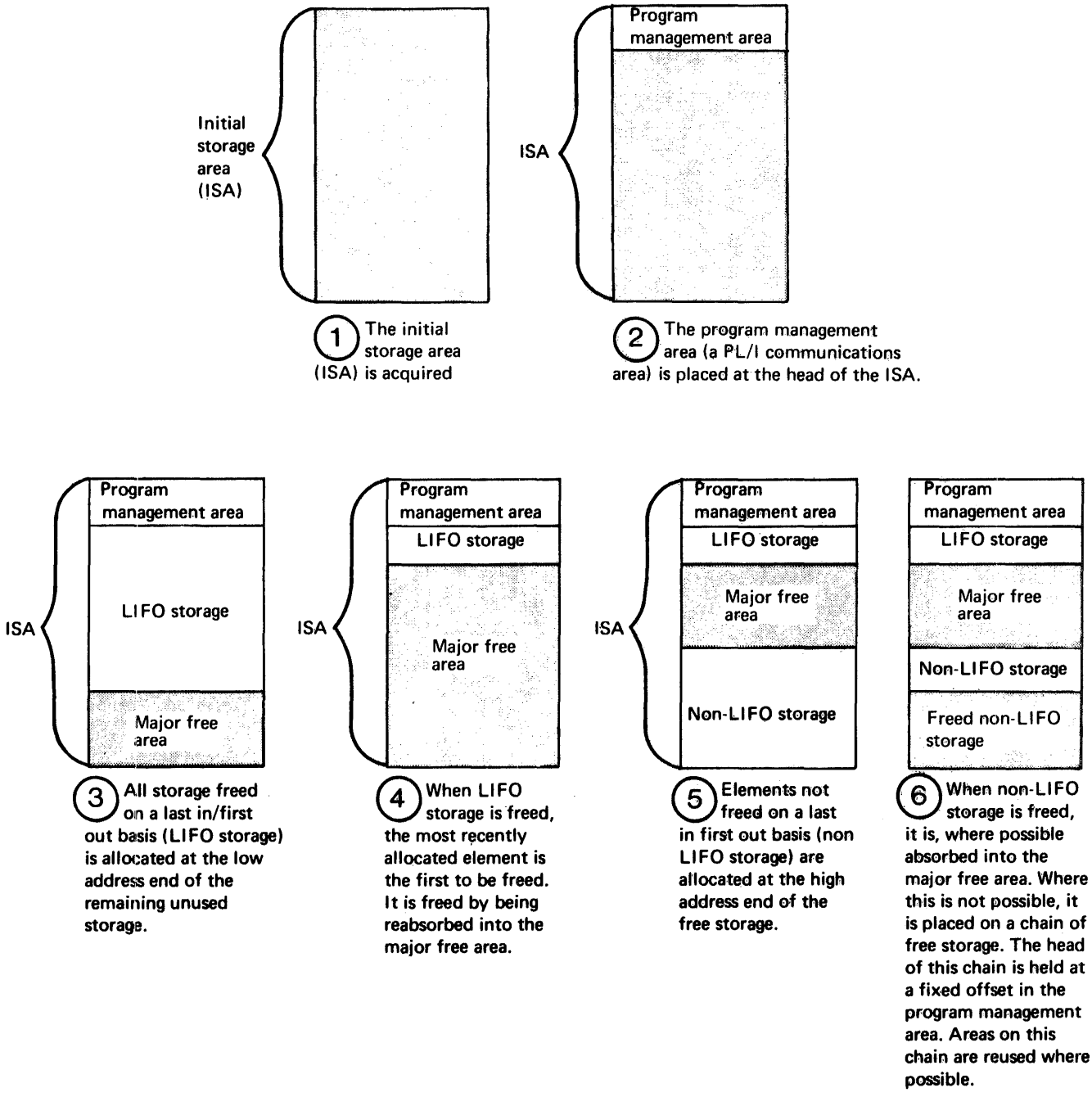


Figure 1.2. Use of PL/I dynamic storage

requires a special arrangement of control blocks and registers for correct execution. This arrangement of control blocks and registers is known as the PL/I environment. Execution consequently becomes a three-stage process:

1. Setting-up the environment. This is handled by the PL/I initialization routines IBMBPIR and IBMBPII.

2. Executing the program.

3. Completing jobs after execution. This consists of closing any files that are left open and returning control either to the supervisor or to a calling module. It is handled by a return to the initialization routine which calls a termination routine.

Factors Affecting Implementation

Three major factors influence the design of the executable programs produced by the optimizing compiler. These factors are inherent in the language, and are:

1. The modular structure of PL/I programs

   The PL/I language allows the programmer to divide his program into a series of blocks that can be written and compiled independently of each other.

2. The dynamic allocation and freeing of storage

   Automatic, controlled, and based variables all have their storage allocated and freed dynamically. This implies a system of re-use of storage to reduce space requirements.

3. The comprehensive facilities offered by the PL/I language

   The PL/I language offers more facilities than any other high-level language. These facilities include allowing the PL/I program to control the flow of execution after any PL/I interrupt.

Key Features of the Executable Program

Taken together, the factors outlined above are responsible for the main features of the executable program produced by the compiler. These features are:

1. A communications area addressed by a dedicated register throughout the execution of the program.

2. A scheme to handle dynamic storage allocation.

3. The use of standard subroutines from the PL/I libraries, to handle such standard tasks as the housekeeping scheme and error handling.

4. The use of initialization routines to set up the communications area and initiate the housekeeping scheme. All PL/I modules are compiled on the assumption that the initialization routines have been called before they are entered.

5. The issuing, by the initialization routines, of SPIE and STAE macro instructions to trap interrupts and ABENDs, and allow them to be handled as defined by PL/I.

These features are discussed further below.

COMMUNICATIONS AREA

The facilities offered by the PL/I language, particularly the error-handling facilities, imply that certain items must be accessible at all times during execution. To simplify accessing such items, a standard communications area is set up for the duration of execution. This area is known as the task communications area (TCA), and is addressed by register 12 throughout execution.

DYNAMIC STORAGE ALLOCATION

The principles of the dynamic storage scheme are illustrated in figure 1.2.

The allocation and freeing of automatic storage on a block-by-block basis implies an automatic facility for the re-use of such storage. This problem and the problem of inter-block communication are solved by having, for each block, a save area that contains register save information, automatic variables, and housekeeping information. This area is known as a dynamic storage area (DSA). It consists of the standard operating system save area concatenated with certain housekeeping information and with storage for automatic variables. DSAs are held contiguously in a last-in/first-out (LIFO) storage stack and are freed and allocated by the alteration

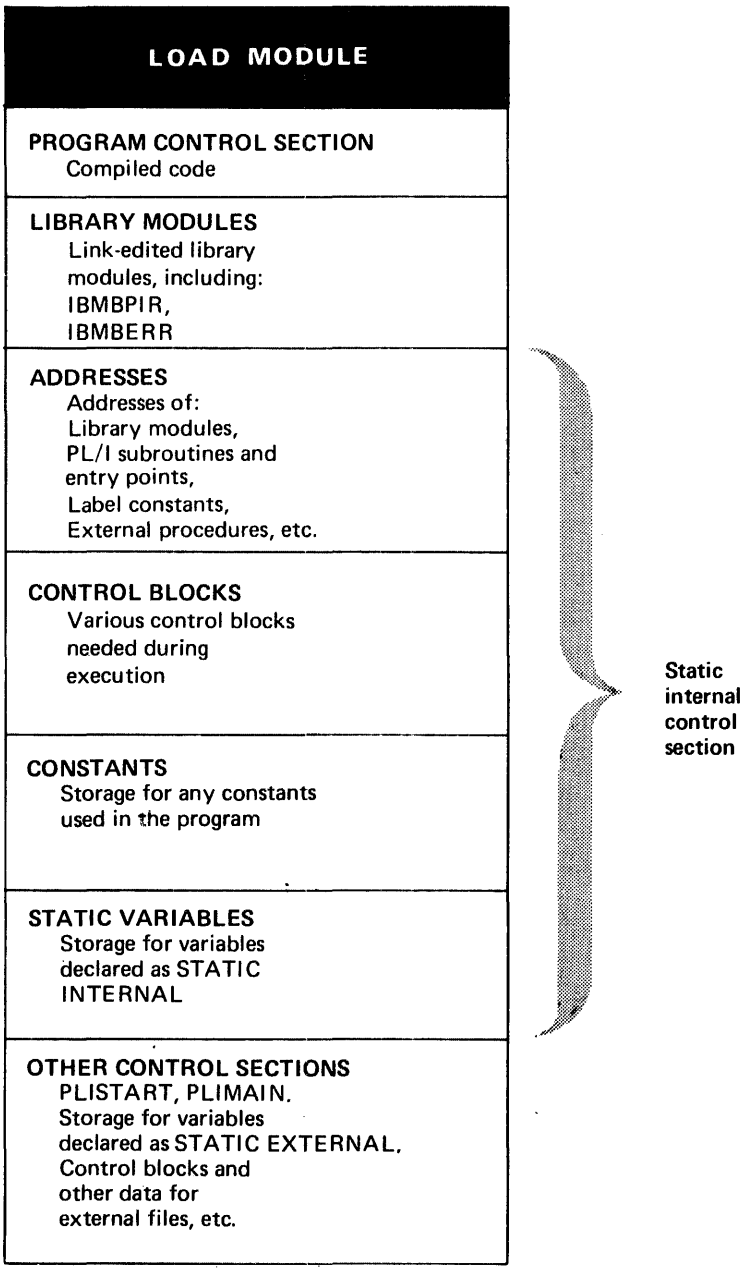


Figure 1.3. Contents of a typical load module

of pointer values.

On entry to a block, the registers of the preceding block are stored in the previous DSA and a new DSA is acquired. A chainback pointer to the previous DSA is placed in the new DSA. This arrangement allows access to information in previous blocks. Register 13 is pointed at the head of the DSA for the current block. The code that carries out this and any other block initialization is known as prologue code. To obviate the need for special coding in the main procedure, a dummy DSA is set up by an initialization routine, and register 13 points at this dummy DSA on entry to the main procedure.

In addition to automatic variables, certain other types of storage are allocated and freed dynamically. Such items as are not freed on a last-in/ first-out basis are kept in a second stack. If items within this stack are freed, they are placed on a free-area chain. The storage scheme is handled partly by compiled code and partly by a resident-library routine. Compiled code acquires and frees space in the LIFO storage stack.

The library routine IBMBPGR is called when non-LIFO dynamic storage has to be allocated or freed, or when there is insufficient space for an allocation of storage in the LIFO stack.

USE OF LIBRARY SUBROUTINES

The use of library subroutines simplifies compilation. On the other hand, using such routines slows execution because they cannot be tailored for the particular situation in hand, and because they incur the overhead of saving and restoring registers. Library subroutines are used for handling standard jobs such as program initialization and error handling, and for those items that require interpretive code. Interpretive code is required when a significant part of the data will not be available until execution.

Two PL/I libraries are used by the OS PL/I Optimizing Compiler: the OS PL/I Resident Library and the OS PL/I Transient Library. Transient library routines have the advantage of saving space, because they require storage only when they are actually in use. Resident library routines, however, have the advantage of speed, because they do not have to be loaded during execution of the PL/I program. Dividing subroutines into transient and resident types enables the compiler to balance the advantages of both types and so to produce programs that combine fast execution with reduced space overheads.

INITIALIZATION/TERMINATION ROUTINES

The job of the initialization routines is to prepare a standard environment for all procedures compiled by the OS PL/I Optimizing Compiler. This consists of setting-up the TCA and initializing the storage scheme. A SPIE macro instruction is issued so that all program checks will be intercepted by the PL/I error-handling facilities. A STAE macro instruction is issued to trap ABENDS. On completion of the main procedure control is returned to the initialization routine by the epilogue code of the main procedure. The program is terminated under the control of the initialization routine. Using standard library routines for these tasks reduces the amount of special-case coding that is needed for a main procedure. A consequence is that subroutines can be compiled and tested individually and then joined with other procedures and run without recompilation. If this is done, care must be taken that the main procedure is the first passed to the linkage editor.

Note: Use of the linkage editor ENTRY statement will not have the desired results as the program must be entered via the initialization routine.

Contents of a Typical Load Module

The contents of a typical load module are shown in figure 1.3. The contents are:

1. Compiled code (the executable machine instructions that have been generated).

2. Link-edited routines. These will include resident library routines and probably OS data management routines. Certain resident library routines are included in every executable program phase. These are the initialization routine, IBMBPIR, and the error handler, IBMBERR. Other resident routines are included as required.

As well as executable machine instructions, the program requires certain control information and addresses. Some of these are listed in figure 1.3, but full details are given in chapter 2. The figure also shows PLISTART, which passes control to the initialization routine, and PLIMAIN, which holds the address of the start of

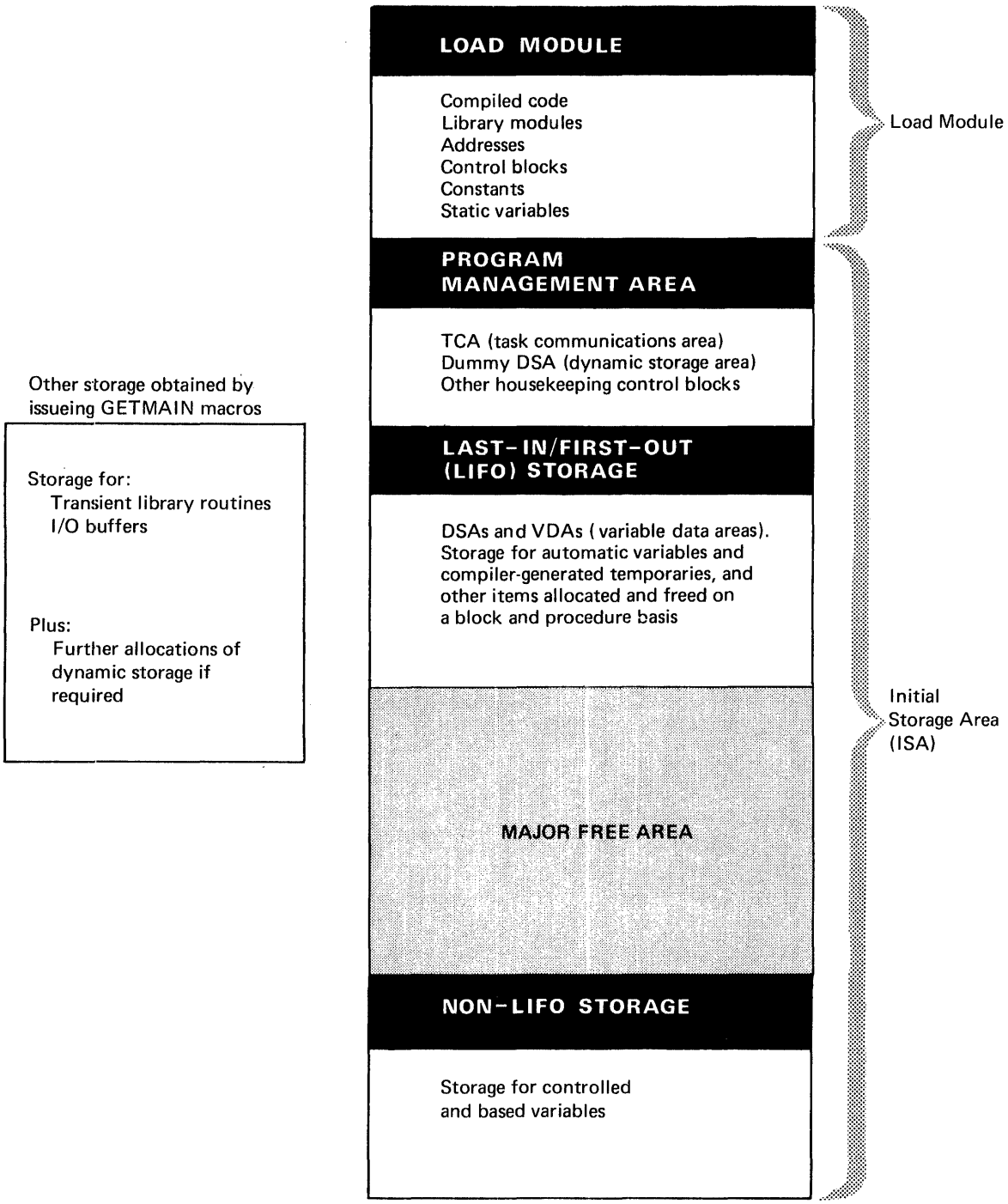


Figure 1.4. Use of storage

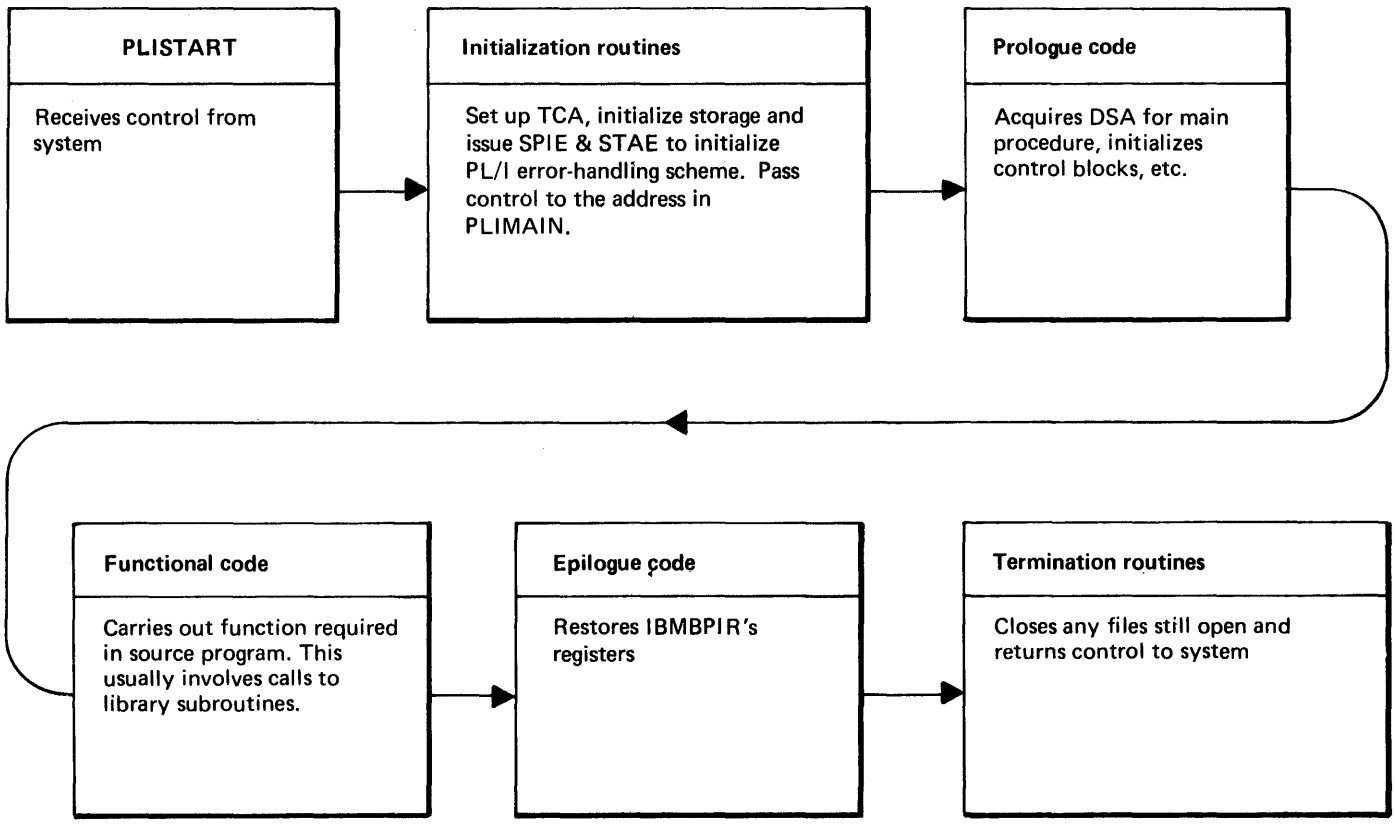


Figure 1.5. Flow of control during execution

compiled code.

The Overall Use of Storage

The overall use of storage is illustrated in figure 1.4. As can be seen, an area known as the initial storage area (ISA) is acquired for program management and PL/I dynamic storage. The program management area is set up by the initialization routines, and includes the TCA and the dummy DSA discussed above. The remainder of the ISA is used for PL/I dynamic storage allocations. The LIFO stack starts beyond the end of the program management area and expands, as necessary, towards the end of the ISA. Non-LIFO dynamic storage starts at the end of the ISA and expands towards the LIFO stack. Storage for I/O buffers and transient library routines is acquired by issuing GETMAIN macro instructions.

The Process of Execution

The process of execution is illustrated in figure 1.5. The processes involved for a sample program are described below.

```
SAMPLE:  PROC OPTIONS(MAIN);
   INPUT:  GET LIST(Y,Z);
             .
             .
             .
     (process data as required)
             .
             .
             .
           PUT LIST(X);
           IF X<500 THEN GO TO INPUT;
           END;
```

During execution:

1. The control program passes control to the control section PLISTART, which has been generated by the compiler.

2. PLISTART calls the resident library initialization routine, IBMBPIR.

3. IBMBPIR and IBMBPII(called by IBMBPIR) set up the PL/I environment. IBMBPIR then passes control to the main procedure compiled code, with register 12 pointing at the TCA and register 13 pointing at the dummy DSA. The address to which IBMBPIR passes control is held in the control section PLIMAIN.

4. Compiled code prologue stores the contents of the registers used by IBMBPIR in the dummy DSA and acquires a DSA for the main procedure.

5. Compiled code calls the library routines used for stream I/O. These in turn call transient routines to open the standard files and further transient routines to interface with data management routines.

6. Processing is then carried out by compiled code. Further calls to the library may be involved if, for example, mathematical functions are used.

7. The stream output will involve further steps similar to those described in 5, above.

8. When the END statement is reached, the epilogue code is entered. This restores the registers of IBMBPIR, the initialization routine, and returns control to IBMBPIR.

9. IBMBPIR raises the FINISH condition, calling the resident error-handling module IBMBERR, which searches for a FINISH on-unit. Finding none, it calls IBMBPIT. IBMBPIT carries out certain housekeeping tasks, including calling IBMBOCL to close the files that have been opened. IBMBPIT returns control to IBMBPIR, which returns control to the supervisor.

This program illustrates the main points mentioned earlier in the chapter. The initialization routines are used in steps 3 and 9, to set up and discard the PL/I environment. The storage management scheme is illustrated in the prologue and epilogue code in steps 4 and 8. The communications area (TCA) is set up by the initialization routine, and the use of standard library subroutines is shown in steps 5 and 7. The use of special error and PL/I condition handling code is shown in step 9.

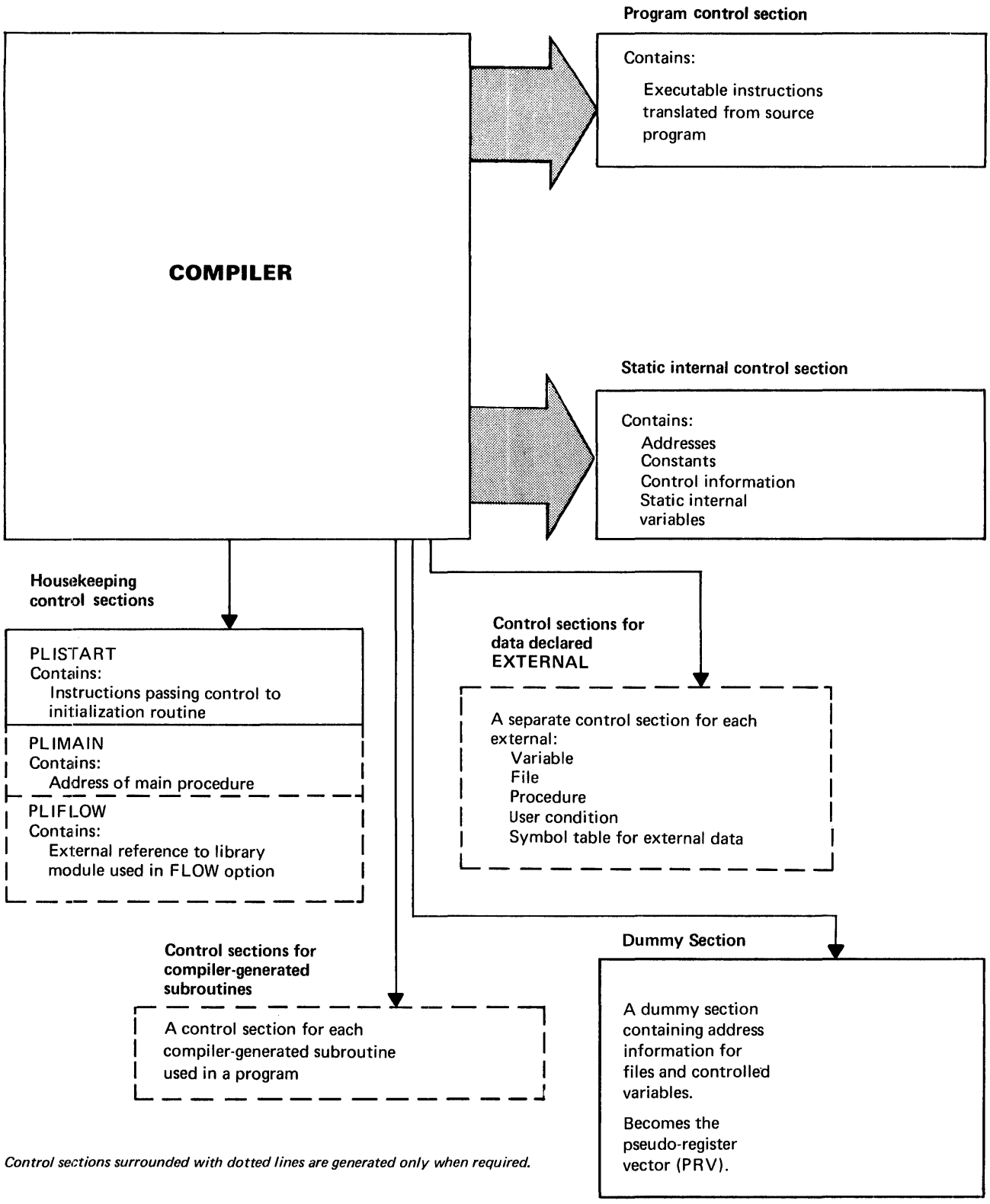


Figure 2.1. The output from the compiler

Chapter 2: Compiler Output

Introduction

This chapter describes that part of the load module that is generated by the compiler. The compiler output is a relocatable object module consisting of a series of records in card-image format. These records contain either machine instructions, constants, or external or internal addresses to be resolved by the linkage editor. The records are known as:

TXT records - records containing machine instructions or constants.

RLD records - records containing internal addresses.

ESD records - records containing external addresses.

Further information about the output passed to the linkage editor is given in the publication OS PL/I Optimizing Compiler: Program Logic.

There are two main control sections output by the compiler. These are:

1. The program control section, holding the executable instructions translated from the PL/I program.
2. The static internal control section holding constants, addresses, and static variables.

A number of other control sections are also generated. These either handle certain housekeeping functions, or are used for external data which may have identical control sections generated for it by other compilations.

Workspace and storage for automatic variables is acquired during execution, normally by the prologue code that is executed at the start of every block.

The output from the compiler is shown in figure 2.1 and listed below:

1. Control sections that are always generated

Program control section	Containing executable instructions.
Static internal control section	Containing addresses, control blocks, constants, and STATIC INTERNAL variables.
PLISTART	The entry point for the executable program phase. Passes control to initialization routine.

2. Control sections that are generated only when required

PLIMAIN	Containing the address of the entry point of the main procedure. (Generated only for procedures with OPTIONS(MAIN).)
PLIFLOW	A control section generated when the compiler FLOW option is specified. (See chapter 7.)
PLICOUNT	A control section generated when the COUNT compiler option is specified.
Static external control sections	A static external control section is generated for every external variable, file, and procedure.
Plus control sections for	Each user-defined condition, and each compiler-generated subroutine used.

3. Dummy sections

Pseudo-register vector	A dummy section used in addressing files and controlled variables.

The remainder of this chapter deals with these items in further detail. Where possible, it refers to the object program listing, because this is the form in which the output from the compiler is most readily accessible.

The two control sections, PLISTART and

Name	Contents	Compiler Option
Source program	Source program statements	SOURCE
Aggregate table	Names and storage requirements of structures and arrays	AGGREGATE
Storage requirements	Names and storage requirements of all procedures	STORAGE
ESD references	Name, type, and identifier of all external references generated by the compiler*	ESD
Statistics	Number of source records, program text statements, and object code bytes	ESD
Static storage	Contents of static internal and static external control sections in hexadecimal notation with comments	MAP
Table of offset and statement number	Offsets, within code, of the start of each statement	OFFSET
Object program	The contents of the program control section in hexadecimal and translated into a pseudo-assembler-language format	LIST
Variables offset MAP	The offsets of automatic and static internal variables from their defining base	MAP

* External references within library modules are not included.

Figure 2.2. Contents of listing and associated compiler options

PLIMAIN, are used during program initialization. PLISTART holds the address of the library initialization routine IBMBPIR, which will be entered at the start of the program. PLIMAIN holds the address of the start of the code for the main procedure. This is the address to which the library initialization routine branches when initialization is complete; it is marked "*REAL ENTRY" in the object-program listing.

A PLIMAIN control section is generated for every procedure for which OPTIONS (MAIN) is specified in the procedure statement. When two such procedures are being run together, control will pass to the first of the procedures processed by the linkage editor (unless the program's JCL specifically indicates otherwise).

The format of PLIMAIN and PLISTART is given in appendix A.

If the compiler FLOW option is being used, a control section called PLIFLOW is also generated. This contains code that results in the link-editing of the trace module IBMBEFL and also contains the values of "n" and "m" specified in the option. The format of PLIFLOW is given in chapter 7.

The Organization of this Chapter

The remainder of this chapter describes the contents of the static internal control section and the program control section. First the conventions used in the object program listing and the static storage map are described. Descriptions of the two control sections follow. The description of the program control section covers the conventions used in the object program code such as register usage, method of handling flow of control, and addressing information. The chapter is completed by a short discussion of the effects of optimization.

Listing Conventions

Figure 2.2 shows all the program listing information that can be produced by the

```
                        SOURCE

1           EXAMPLE: PROC OPTIONS (MAIN) REORDER;
2    1                 DCL X(10),Y,Z INITIAL (0);

3    1                 GET EDIT(X,Y)(F(3),X(11));

4    1                     DO I=1 TO Y;
5    1  1                  Z=Z*X(I);
6    1  1                  END;
7    1                 PUT EDIT(Z)(A);
8    1               END;
```

```
      STATIC INTERNAL STORAGE MAP                          STATIC EXTERNAL CSECTS

000000  00000008              PROGRAM ADCON          000000  0000000000000000      DCICB
000004  0000005E              PROGRAM ADCON                  0000000000000000
000008  00000068              PROGRAM ADCON                  0F4700140005E2E8
00000C  00000000              A..IELCGIA                     E2C9D520
000010  00000000              A..IELCGIB
000014  00000000              A..IELCGOA
000018  00000000              A..IELCGOB             000000  FFFFFFFC41201000      DCICB
00001C  00000000              A..IBMBCACA                    02D70F0000000000
000020  00000000              A..IBMBCEDB                    FF6000140008E2E8
000024  00000000              A..IBMBCHFD                    E2D7D9C9D5E3008B
000028  00000000              A..IBMBCTHD
00002C  00000000              A..IBMBCVDY
000030  00000000              A..IBMBOCIA
000034  00000000              A..IBMBOCLC
000038  00000000              A..IBMBSACA
00003C  00000000              A..IBMBSEIA
000040  00000000              A..IBMBSEIT
000044  00000000              A..IBMBSEOA
000048  00000000              A..IBMBSFIA
00004C  00000000              A..IBMBSIIA
000050  00000000              A..IBMBSICA
000054  00000000              A..IBMBSIOT
000058  00000000              A..IBMBSXCA
00005C  00000000              A..STATIC
000060  08040680              DED..X
000064  500000030080          FED
00006A  6000000B              FED
00006E  5800000C              FED
000072  000A                  CONSTANT
000074  0001                  CONSTANT
000076  0004                  CONSTANT
000078  91E091E0              CONSTANT
00007C  00000000              CONSTANT
000080  46008000              CONSTANT
000084  41100000              CONSTANT
000088  00000000              A..DCLCB
00008C  00000000              A..DCICB
000090  00000000              A..DCLCB
000094  80000000              A..TEMP
000098  00000000              A..DCLCB
00009C  80000000              A..TEMP
0000A0  0000010400000068      COMPILER LABEL CL.9
```

Figure 2.3. Example of static storage listing

```
000004  91 40 1 010              TM     16(1),X'40'
000008  47 10 7 052              BO     **+74
00000C  58 F0 1 014              L      15,20(0,1)
000010  50 70 1 01C              ST     7,28(0,1)
000014  58 70 1 00C              L      7,12(0,1)
000018  48 E0 F 050              LH     14,80(0,15)
00001C  4B E0 7 002              SH     14,2(0,7)
000020  40 E0 F 050              STH    14,80(0,15)
000024  58 E0 F 04C              L      14,76(0,15)
000028  4A E0 7 002              AH     14,2(0,7)
00002C  50 E0 F 04C              ST     14,76(0,15)
000030  48 E0 1 020              LH     14,32(0,1)
000034  41 E0 E 001              LA     14,1(0,14)
000038  40 E0 1 020              STH    14,32(0,1)
00003C  40 E0 F 052              STH    14,82(0,15)
000040  91 10 1 010              TM     16(1),X'10'
000044  07 86                    BCR    8,6
000046  58 70 1 01C              L      7,28(0,1)
00004A  58 F0 7 068              L      15,104(0,7)
00004E  05 EF                    BALR   14,15
000050  07 F6                    BR     6
000052  58 F0 7 06C              L      15,108(0,7)
000056  05 EF                    BALR   14,15
000058  58 E0 1 008              L      14,8(0,1)
00005C  50 E0 D 04C              ST     14,76(0,13)
000060  94 BF 1 010              NI     16(1),X'BF'
000064  07 F6                    BR     6
000066  07 00                    NOPR   0
000068                           DC     AL4(0)
00006C                           DC     AL4(0)

* END OF COMPILER GENERATED SUBROUTINE


* STATEMENT NUMBER  1
000000                           DC     C'EXAMPLE'
000007                           DC     AL1(7)

* PROCEDURE                              EXAMPLE

* REAL ENTRY
000008  90 EC D 00C              STM    14,12,12(13)
00000C  47 F0 F 014              B      **+16
000010  00000000                 DC     A(STMT. NO. TABLE)
000014  00000130                 DC     F'304'
000018  00000000                 DC     A(STATIC CSECT)
00001C  58 30 F 010              L      3,16(0,15)
000020  58 10 D 04C              L      1,76(0,13)
000024  58 00 F 00C              L      0,12(0,15)
000028  1E 01                    ALR    0,1
00002A  55 00 C 00C              CL     0,12(0,12)
00002E  47 D0 F 030              BNH    **+10
000032  58 F0 C 074              L      15,116(0,12)
000036  05 EF                    BALR   14,15
000038  58 E0 D 048              L      14,72(0,13)
00003C  18 F0                    LR     15,0
00003E  90 E0 1 048              STM    14,0,72(1)
000042  50 D0 1 004              ST     13,4(0,1)
000046  41 D1 0 000              LA     13,0(1,0)
00004A  50 50 D 058              ST     5,88(0,13)
00004E  92 80 D 000              MVI    0(13),X'80'
000052  92 24 D 001              MVI    1(13),X'24'
000056  D2 03 D 054 3 078        MVC    84(4,13),120(3)
00005C  05 20                    BALR   2,0

* PROLOGUE BASE

* INITIALISATION CODE FOR Z
00005E  78 40 3 07C              LE     4,124(0,3)
000062  70 40 D 0AC              STE    4,Z
000066  05 20                    BALR   2,0

* PROCEDURE BASE


* STATEMENT NUMBER  3
000068  41 90 D 108              LA     9,264(0,13)
00006C  50 90 3 094              ST     9,148(0,3)
000070  96 80 3 094              CI     148(3),X'80'
000074  92 24 D 119              MVI    281(13),X'24'
000078  41 E0 3 0A0              LA     14,160(0,3)
00007C  50 E0 D 120              ST     14,288(0,13)
000080  41 10 3 090              LA     1,144(0,3)
000084  58 F0 3 04C              L      15,A..IBMBSIIA
000088  05 EF                    BALR   14,15
00008A  41 A0 2 06E              LA     10,CL.8
00008E  48 E0 3 074              LH     14,116(0,3)
000092  50 E0 D 0E8              ST     14,232(0,13)
000096                  CL.5     EQU    *
000096  58 90 D 0E8              L      9,232(0,13)
00009A  8E 90 0 002              SIA    9,2
00009E  41 E9 D 0B4              LA     14,VC..X(9)
0000A2  41 F0 3 060              LA     15,DED..VO..X
0000A6  41 10 D 108              LA     1,264(0,13)
0000AA  50 10 D 0EC              ST     1,236(0,13)
0000AE  90 EF 1 008              STM    14,15,8(1)
```



```
0000B2  05 AA                    BALR   10,10
0000B4  58 E0 D 0E8              L      14,232(0,13)
0000B8  4A E0 3 074              AH     14,116(0,3)
0000BC  50 E0 D 0E8              ST     14,232(0,13)
0000C0  49 E0 3 072              CH     14,114(0,3)
0000C4  47 C0 2 02E              BNH    CL.5
0000C8  41 E0 D 0A8              LA     14,Y
0000CC  50 E0 1 008              ST     14,8(0,1)
0000D0  05 AA                    BALR   10,10
0000D2  47 F0 2 09C              B      CL.9
0000D6                  CL.8     EQU    *
0000D6  41 E0 3 064              LA     14,100(0,3)
0000DA  58 10 D 0EC              L      1,236(0,13)
0000DE  58 70 3 00C              L      7,A..IELCGIA
0000E2  05 67                    BALR   6,7
0000E4  58 F0 3 048              L      15,A..IBMBSFIA
0000E8  05 EF                    BALR   14,15
0000EA  58 70 3 010              L      7,A..IELCGIB
0000EE  05 67                    BALR   6,7
0000F0  05 AA                    BALR   10,10
0000F2  41 E0 3 06A              LA     14,106(0,3)
0000F6  58 10 D 0EC              L      1,236(0,13)
0000FA  58 70 3 00C              L      7,A..IELCGIA
0000FE  05 67                    BALR   6,7
000100  47 F0 2 06E              B      CL.8
000104                  CL.9     EQU    *
00013E  50 E0 D 0F8              ST     14,248(0,13)

* CALCULATION OF COMMONED EXPRESSION FOLLOWS
000142  78 20 D 0F4              LE     2,244(0,13)
000146  70 20 D 0FC              STE    2,252(0,13)

* END OF COMMON CODE

* CONTINUATION OF STATEMENT NUMBER  4
00014A                  CL.2     EQU    *


* STATEMENT NUMBER  5
00014A  58 70 D 0F8              L      7,248(0,13)
00014E  78 40 D 0AC              LE     4,Z
000152  7C 47 D 0B4              ME     4,VC..X(7)
000156  70 40 D 0AC              STE    4,Z


* STATEMENT NUMBER  6

* METHOD OR ORDER OF CALCULATING EXPRESSIONS CHANGED

* CODE MOVED FROM STATEMENT NUMBER 4
00015A  78 00 D 0FC              LE     0,252(0,13)
00015E  7A 00 3 084              AE     0,132(0,3)
000162  70 00 D 0FC              STE    0,252(0,13)
```

Figure 2.4. Part of an object program listing (For source see Figure 2.3)

compiler. It also shows the relevant compiler options and summarizes the information that will be produced if these options are specified. Some or all of these options may be deleted at system generation time. To obtain deleted options, the correct password (specified at system generation time) must be specified in the CONTROL option.

This chapter describes the contents of the static-storage map and the object-program listing. Information on the other items generated is given in the publication OS PL/I Optimizing Compiler: Programmer's Guide.

STATIC-STORAGE MAP

The static-storage map is a formatted listing of the contents of the static internal and static external control sections. The static control sections contain items grouped in the following order:

1. Address constants for entry points to procedures, and for branch instructions.
2. Address constants for resident library subroutines.
3. Address constants for addressing static storage beyond 4K.
4. The constants pool, which contains source program constants, data element descriptors, locator/descriptors, symbol tables, declare control blocks (DCLCBs), and other control blocks.
5. Static variables.

The constants pool and the static-variable sections of static storage begin on doubleword boundaries.

The static control section is listed, each line comprising the following elements:

1. Six-digit hexadecimal offset.
2. Hexadecimal text, in 8-byte sections where possible.
3. Comment, indicating the type of item to which the text refers; a comment appears against only the first line of the text for an item.

A typical static listing is shown in figure 2.3.

The following comments are used (xxx indicates the presence of an identifier):

A.. - Address constant.

COMPILER LABEL CL.nn - Compiler-generated label followed by CL plus number.

CONDITION CSECT - Control section for programmer-named condition

CONSTANT

CSECT FOR EXTERNAL VARIABLE - Control section for external variable.

D.. - Descriptor.

DED.. - Data element descriptor.

ENVB - Environment control block.

DCLCB - Declare control block.

FED.. - Format element descriptor.

KD.. - Key descriptor.

ONCB - ON control block.

PICTURED DED.. - Pictured DED.

RD.. - Record descriptor.

SYMTAB - Symbol table.

USER LABEL xxx - Source program label for xxx.

xxx - Name of variable. If the variable is not initialized, no text appears against the comment; there is also no static offset if the variable is an array. (The static offset can be calculated from the array descriptor if required.)

OBJECT-PROGRAM LISTING

By including the option LIST in the PROCESS statement, the programmer can obtain a listing of the compiled code, known as the object-program listing. This listing consists of the machine instructions plus a translation of these instructions into a form that resembles assembler language, and a number of comments such as the statement number. The format of this listing is shown in figure 2.4. As can be seen, blocks of code are headed by the number of the statement in the PL/I program to which they are equivalent. When optimization has resulted in code being moved out of a statement, this is indicated. Only executable statements appear in the listing. DECLARE statements are not

included, because they have no direct machine-code equivalent. To simplify understanding of the listing, the names of PL/I variables are inserted, rather than the addresses that appear in the machine code. Special mnemonics are used when referring to control blocks and other items.

Statements in the object program listing are ordered by block. Statements in the outermost block are given first, followed by statements in the inner blocks. Thus the order of statements will frequently differ from that of the source program.

Every object-program listing begins with the name of the procedure. The name is defined as a constant in a DC instruction. This is followed by another constant containing the length of the procedure name. Then comes the name of the procedure, as a comment, followed by code under the heading "REAL ENTRY." This is the point at which the code will, in fact, be entered. The second section of code is the prologue, which carries out various housekeeping tasks and is described more fully later in this chapter. The end of the prologue is marked by the message "PROCEDURE BASE." This is followed by a translation of the first executable statement in the PL/I source program.

The comments used in the listing are as follows:

* PROCEDURE xxx - identifies the start of the procedure labeled xxx.

* REAL ENTRY xxx - heads the initialization code for an entry point to a procedure labeled xxx.

* PROLOGUE BASE - identifies the start of the prologue code common to all entry points into that procedure.

* PROCEDURE BASE - identifies the address loaded into the base register for the procedure.

* STATEMENT LABEL xxx - identifies the position of source program statement label xxx

* PROGRAM ADDRESSABILITY. REGION BASE - identifies address to which the program base is updated if the program size exceeds 4096 bytes and consequently cannot be addressed from one base.

* CONTINUATION OF PREVIOUS REGION - identifies the point at which addressing from the previous program base recommences.

* END OF COMMON CODE - identifies the end of code used in the execution of more than one statement.

* END PROCEDURE xxx - identifies the end of the procedure labeled xxx.

* BEGIN BLOCK xxx - indicates the start of the begin block with label xxx.

* END BLOCK xxx - indicates the end of the begin block with label xxx.

* BEGIN BLOCK - GENERATED NAME BLOCK.nn - indicates the start of an unnamed begin block for which the compiler has generated the name BLOCK.nn, where nn is two hexadecimal digits.

* END BLOCK.nn - indicates the end of the begin block with compiler-generated name BLOCK.nn.

* STATEMENT NUMBER n - identifies the start of code generated for statement number n in the source listing.

* INTERLANGUAGE PROCEDURE xxx - identifies the start of encompassing procedure xxx (see chapter 13).

* END INTERLANGUAGE PROCEDURE xxx - identifies the end of encompassing procedure xxx. (See chapter 13)

* COMPILER GENERATED SUBROUTINE xxx - indicates the start of compiler-generated subroutine xxx.

* END OF COMPILER GENERATED SUBROUTINE - indicates the end of the compiler-generated subroutine.

* ON UNIT BLOCK - indicates the start of an on-unit block.

* ON UNIT BLOCK END - indicates the end of the on-unit block.

* END PROGRAM - indicates the end of the external procedure.

* INITIALIZATION CODE FOR OPTIMIZED LOOP FOLLOWS - indicates that some of the code that follows has been moved from within a loop by the optimization process.

* CODE MOVED FROM STATEMENT NUMBER n - indicates object code moved by the optimization process to a different part of the program and gives the number of the statement from which it originated.

* CALCULATION OF COMMONED EXPRESSION FOLLOWS - indicates that an expression used more than once in the program is calculated at this point.

REGISTER USAGE (compiled code)				
	Dedicated registers	Work registers (plus special use)	Preferred registers	Notes
0		General		Cannot be used as base
1		General + address of parameter list		
2	Address of program base			Saved during in-line record I/O and TRT instructions
3	Address of static base			
4				Address of temporary base if DSA size greater than 3896 bytes
5		General + static chainback on entry to procedure	Preferred register for DO loop control variable	
6		General		
7		General		
8		General		
9		General		
10		General	Preferred registers for DO loop control when BXLE instruction is used	
11		General		
12	Address of TCA			
13	Address of current DSA			
14		General + branch-and-link to library and other routines		
15				

Figure 2.5. Register usage in compiled code

* METHOD OR ORDER OF CALCULATING EXPRESSIONS CHANGED - indicates that the order of the code following has been changed to optimize the object code.

In certain cases, mnemonics are used to identify the type of operand in an instruction, and, where applicable, this is followed by the name of a PL/I variable. The following prefixes are used:

A..	Address constant.
ADD..	Aggregate descriptor descriptor.
BASE..	Base address of a variable.
BLOCK.nn	Label created for an otherwise unlabeled block.
CL.nn	Compiler-generated label.

D..	Descriptor.
DED..	Data element descriptor.
WSP.n	Workspace, followed by decimal number of the block of allocated workspace.
L..	Length of variable.
LOCATOR..	Locator.
RKD..	Record or key descriptor.
VO..	Virtual origin (the address where element 0 would be held for a one-dimensional array, element 0,0 for a two-dimensional array, etc.).

Static Internal Control Section

The static internal control section contains the majority of items that are not executable instructions. The contents of a typical static control section are shown in figure 2.3.

The first part of the static internal control section contains addresses. These are held in the order:

1. Addresses of library modules
2. Addresses of entry points
3. Addresses of label constants that may be assigned to label variables
4. Addresses of external procedures (other than library modules)

The address section is followed by a section known as the constants pool. This contains the following items (if required by the program):

Constants	Constant values used by compiled code.
ONCBs	Control blocks used in error handling. (See chapter 7.)
Descriptors, locators & DEDs (data element descriptors)	Control information used by compiled code and library. (See chapter 4.)
Symbol table address vector	Control information used in data-directed I/O. (See chapter 4.)
Diagnostic statement table	Information on statement numbers.

Items are arranged according to their alignment requirements, those requiring doubleword alignment first, followed by fullword, halfword, byte, and bit.

The final section of the static internal control section holds the static variables. These are held in size order, smallest first, as for automatic variables: first the variables of 8 bytes or less, next the variables of 2048 bytes or less, and finally any variable greater than 2048 bytes. This system ensures that the smallest possible number of items will require indirect addressing, since it will always be the largest variables that overflow the 4K boundary. Within each division, items are grouped according to alignment stringencies, starting with those requiring doubleword alignment. This method ensures optimum use of storage.

Program Control Section

The program control section contains the executable instructions that are a translation of the PL/I source program. The format of each program control section depends on the contents of the source program. The discussion that follows covers items that will be common to all source programs.

To keep discussions of subjects as complete as possible the chapter also includes descriptions of certain library functions when they are closely allied with the subject under discussion.

REGISTER USAGE

Details of register usage during the execution of compiled code are given in figure 2.5.

Four general registers are used as bases for addressing various types of data; these are known as dedicated registers. The remainder of the registers are used as they are required and are known as work registers

Dedicated registers are:

R2 Program base.

R3 Static base.

R12 TCA pointer.

R13 DSA pointer.

This arrangement of dedicated registers allows compiled code the use of five even/odd work register pairs. These are (0,1), (6,7), (8,9), (10,11), and (14,15).

Certain registers have special tasks for which they are always used, or for which they are preferred and used when available. These tasks are shown in figure 2.5.

Dedicated Registers

Register 2 - Program Base Register: Register 2 is the program base register and is used for branching within the code. When the code exceeds 4K, register 2 is updated so that all branching is done on this register. During in-line I/O (when data management calls are handled by compiled code rather than by library subroutines), and during the execution of TRT instructions, the program base register contents are saved and the register used for other purposes.

Register 3 - Static Base Register: Register 3 points to the start of the static internal control section. The items to be found in this control section in any particular program are listed in the static-storage map put out by the compiler. (See "Static Internal Control Section," later in this chapter.) When the static control section is larger than 4K bytes, a further base register is used.

Register 12 - TCA: Offsets from register 12 are used to address the various fields in the TCA. The TCA is discussed further in chapter 5. Its format is shown in appendix A.

Register 13 - Current DSA: Register 13 points to the current DSA and is used to address the automatic variables declared in the current procedure or block. References to offsets from register 13 which do not appear as names in the assembler language listing are references to the housekeeping fields held in every DSA. These are discussed in chapter 6; the format of the housekeeping information in a DSA is given in appendix A.

Register 4: When the DSA is larger than 3896 bytes register 4 is used as a base for compiler generated temporaries.

Work Registers

Special or preferred uses for work registers are shown in figure 2.5. Special uses are those for which the register is freed and always used. Preferred uses are those for which the register is used when possible.

Floating-Point Registers

Floating-point registers are all used as general work registers for floating-point data.

Library Register Usage

Register usage in library modules is different from that in compiled code. It is shown in figure 2.6.

In both library and compiled code usage, register 12 points at the TCA, and register 13 at the current DSA. Registers 14 and 15 are used by both library subroutines and compiled code to branch and link between routines.

A further point about library register usage is worth noting. Registers 14 through 4 are normally saved by the library. This is because the majority of library subroutines use only these registers. Consequently, time can be saved by reducing save-restore requirements. However, some library routines also save one or more of registers 5 through 11.

Handling and Addressing Variables and Temporaries

AUTOMATIC VARIABLES

Automatic variables have storage allocated on a procedure or begin-block basis. Variables whose length is known during compilation have storage allocated within the DSA of the block in which they are declared. Variables whose length is not known until execution have their storage allocated in variable data areas (VDAs). VDAs are held in the last-in/first-out storage stack and are acquired in the prologue code after the DSA has been acquired. The same method is used as is used for acquiring the DSA (see above under "Prologue Code.")

Automatic variables, when used in the block in which they are declared, are addressed from register 13, if they are held in the DSA. If they are held in a VDA, a separate base is set up for the VDA and they are addressed from this.

REGISTER USAGE (Library)	
Register	Usage
1	Work register
2	Work register
3	Program base register (dedicated)
4	Work register
5	Work register
6	Work register
7	Work register
8	Work register
9	Work register
10	Work register
11	Work register
12	TCA pointer (dedicated in both library and compiled code)
13	DSA pointer
14	Work register (always used for branch-and-link to other routines)
15	Work register (used with register 14 for branch-and-link)

Figure 2.6. Library register usage

Automatic variables known in any procedure or block are those that are declared in that procedure or block, or in any encompassing procedures or blocks. The method used to address automatic variables in outer blocks is as follows. The address of the DSA of the block in which the required variable was declared is placed in the current DSA. This address can then be accessed from register 13. This is done in the prologue. Frequently, the value is retained in the register used in the initial load and not reloaded when the variable is accessed. Typical code would be

```
L  7,96(0,13)   Pick up address of
                correct DSA

L  8,108(7)     Place value in register
                8
```

COMPILER-GENERATED TEMPORARIES

Because PL/I statements can contain an unlimited number of operands, it is frequently necessary to set up fields containing intermediate results. These fields are known as temporary variables (temporaries) and are allocated within the DSA of the associated block, provided that the size of storage required is known at compile time. Temporaries are addressed from register 13, unless the DSA is longer than 4096 bytes. Because temporary storage is continually being reused, the same offset will not always refer to the same temporary.

Temporaries for Adjustable Variables

Where a temporary is needed to hold a value for an adjustable variable, its size is not predictable until execution. In such cases, a VDA is acquired for the temporary value.

CONTROLLED VARIABLES

Controlled variables are addressed through the pseudo-register vector, as described below under the heading "Pseudo-Register Vector (PRV)". When no allocations of the controlled variable have been made, the PRV offset points to the dummy FCB. Otherwise, it points to the most recent allocation of the controlled variable.

Each controlled variable is headed by a four-word control block that holds the address of the previous allocation (if any), the length of the variable (including the control block), the pseudo-register vector offset, and the task invocation count. The format of this control block is shown in appendix A.

Storage for controlled variables is allocated in non-LIFO storage. If there is room in the ISA, it is allocated within the ISA. Otherwise, a GETMAIN macro instruction is issued to obtain storage.

Stacking and unstacking of controlled variables is handled by a resident library routine, IBMBPAF. IBMBPAF calls on IBMBPGR to obtain and release the storage.

BASED VARIABLES

Based variables are addressed by using the contents of the pointer on which they are based. The pointer is addressed in the usual manner, depending on its storage class.

When a based variable is allocated, a call to the storage management module IBMBPGR is made. IBMBPGR acquires storage in the non-LIFO dynamic storage area and returns the address of the storage in register 1. The address held in register 1 is then placed in the pointer on which the allocated variable is based.

When the variable is freed, a further call to IBMBPGR is made to free the storage.

Pointers: Pointers and offsets are held as fullwords. The null pointer value is X'FF000000'.

STATIC VARIABLES

Static internal variables are held in the static internal control section and are addressed from register 3.

Static external variables are held in separate control sections and are addressed from an address constant in the static internal control section.

ADDRESSING BEYOND THE 4K LIMIT

As described above, variables can, in the simplest case, be addressed by using an offset from one of the base registers. However, as the space required for any particular type of storage can exceed the maximum offset allowed in addressing (4096 bytes), it is necessary to have a scheme to allow addressing of variables beyond this limit.

The method used is to divide storage for automatic variables, temporaries, and static variables into sections of 4096 bytes. The addresses of the second and subsequent sections are then placed in the first section. Addressing of an automatic variable beyond the 4096-byte limit is typically done by code resembling the following:

```
L  6,92(0,13)   Place address of 4K
                boundary in register 6.

AH 7,96(0,6)    Address variable by
                using offset from 4K
                boundary placed in
                register set up in last
                instruction.
```

A similar system is used for addressing any static variables which are at an offset greater than 4096 bytes. The addresses are held in the following areas:

Automatic	Immediately following the housekeeping information of the DSA.
Static	At the head of the first section of static storage.
Temporaries	At the head of temporary storage, following bases of parameters, register save area, and addresses of any outer DSAs.

Constants and variables are held in order of size, with the smallest first. This minimizes the number of items that overflow the 4K boundary.

THE PSEUDO-REGISTER VECTOR (PRV)

Addressing Controlled Variables and Files

In order to address controlled variables and files, a control block known as the pseudo-register vector (PRV) is used. This control block is mapped by the linkage editor as a dummy section with a fullword field for each uniquely named controlled variable or file. During execution, the addresses of the storage allocated to the variables or files are placed in the PRV.

The use of the linkage editor is necessary because controlled variables and files may be external and, consequently, it may be necessary to access them in separately compiled procedures. Other external items are compiled as CSECTs, but this is not possible for files or controlled variables because their associated storage is not allocated until execution. Controlled variables have storage allocated during the execution of an ALLOCATE statement; files are addressed from file control blocks (FCBs), which are created during execution.

References to controlled variables and files are compiled as Q-type address constants. During link-editing, the DXD facility of the linkage editor is used, and the PRV is set up as an external dummy section. Each uniquely named file or controlled variable is allocated an offset within this dummy section, and the Q-type address constants are replaced by this offset.

During compilation

1. Each controlled variable or file reference is compiled as a Q-type address constant that will be used as an offset within the PRV.

2. The compiler generates a DXD instruction for every item requiring pseudo-register addressing.

During link-editing

1. The number of unique names requiring pseudo-register addressing is calculated and placed in a field that can be accessed by a CXD instruction.

2. Each reference to a name generated as a Q-type address constant is replaced by the appropriate offset from the start of the PRV.

During program initialization

1. The length required for the PRV is obtained by use of a CXD instruction. Storage for the PRV is then obtained in the program management area. The address of the PRV is placed in the TCA.

2. The address of the dummy FCB is placed in every field of the PRV.

During execution

1. When storage is allocated to the FCB or controlled variable, the address of the storage is placed in the associated field in the PRV. Comparison with the dummy FCB address can then be made, to determine whether storage has been allocated for the item.

Figure 2.7. Use of the pseudo-register vector (PRV)

Controlled variables and files are addressed via the PRV regardless of whether they are external or internal. The compiler prefixes internal items with the name of their procedure so that their names will be unique. The use of the PRV is summarized in figure 2.7.

The Location of the PRV

The pseudo-register vector is held in the program management area, and is addressed from the TCA.

Whenever a new task is attached, the PRV of the attaching task is copied into the program management area of the attached task. This means that, at the point when the task is attached, the files and controlled variables addressed from the subtask will be the same as those in the parent task. However, because each task has its own PRV, either task may change the addresses without affecting the other.

Initialization of the PRV

To simplify implicit opening of a record I/O file, the PRV is initialized with every field set to point to a control block known as the dummy FCB. Use of this control block as if it were a genuine FCB results in control being passed to the open routines: the file is opened, and a real FCB is created. The address of the real FCB is then placed in the PRV.

Pseudo-register fields for controlled variables are also initialized to point to the dummy FCB, so that the controlled variable allocation mechanism can determine whether an allocation has been made by comparing the PRV value with the address of the dummy FCB. (The address of the dummy FCB is held throughout the program in the TCA, so that the comparison can be made.)

Program Control Data

Program control data comprises pointer, offset, file, area, entry, event, task, and label data.

Pointer and offset data items are each held in a fullword. The data item in both cases consists of an address that is held right-adjusted in the field, padded on the left with zeros. For both data types, the null value is represented by hexadecimal 'FF000000'.

A file variable is held as a fullword containing the address of the declare control block (DCLCB); the DCLCB corresponds to a file constant.

The formats of area, entry, event, task and label data are given in Appendix A.

HANDLING DATA AGGREGATES

PL/I data aggregates are structures and arrays, and include both arrays of structures and structures of arrays.

Array elements are addressed from the virtual origin of an array. This is the point at which the element whose subscripts are all zeros is held, or would be held if no such element is included in the array. Each element can be accessed by using a multiplier for each dimension. The multiplier is the distance between elements in a cross-section of an array. For example, in an array B(9,9) the multiplier for the first dimension is the distance between elements B(1,1) and B(2,1); the multiplier for the second dimension is the distance between elements B(1,1) and B(1,2).

If the bounds of the array and the length of the elements of the array are known during compilation, the values of multipliers can be calculated and placed as constants in the static internal control section. For accessing an element with a constant subscript, the offset from the virtual origin can be calculated during compilation. If the subscript value is a variable, the multiplier must be picked up from static storage during execution and the value calculated.

If the bounds or extents of an array are not known during compilation, a control block known as an array descriptor is set up. This control block is used to hold necessary information about bounds, multipliers, etc. The information is placed in the array descriptor during execution. Array descriptors are described in chapter 4.

Structures are treated in a similar manner. Where all information about a structure is known, it is mapped during compilation and offsets to each item from the start of the structure are known to compiled code. If a structure cannot be mapped during compilation, it is mapped during execution, and the offsets within the structure are placed in a control block known as a structure descriptor. To access an item in the structure, compiled code finds the offsets and calculates the address of each element from them. Structure descriptors and the process of mapping during execution are described in chapter 4.

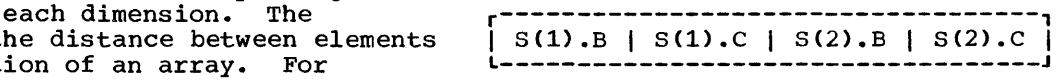
ARRAYS OF STRUCTURES AND STRUCTURES OF ARRAYS

Arrays of structures and structures of arrays are held as they are declared.

The array of structures

```
1 S(2),
  2 B,
  2 C;
```

would be held in the order

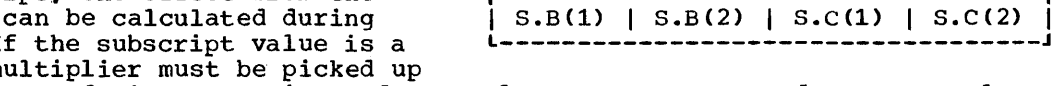


B and C are known as interleaved arrays, because the elements within each array are not contiguous.

The structure of arrays

```
1 S,
  2 B(2),
  2 C(2);
```

would be held in the order



Elements are accessed as array elements in both cases. In the array of structures shown above, both B and C are treated as separate arrays with their own virtual origins and multipliers. The difference would be in the value of the multipliers. When possible, the values of multipliers are calculated during compilation. When adjustable bounds or extents are involved, the necessary data for both arrays of structures and structures of arrays is placed in a structure descriptor (see chapter 4).

ARRAY AND STRUCTURE ASSIGNMENTS

Assignments between structures and arrays of the same format are done by MVC instructions. Provided an array is not interleaved, an assignment will be made to it as a whole, and the elements will not be moved one at a time. Similarly, structures that are contiguous and have the same format are moved as a whole.

```
STM       14,12,12(13)          Store registers of calling program.
BC        *+16                  Branch around constants.
DC        A(STMT NO TABLE)      Constant - address of statement number table.
DC        F'272'                Constant - length required for new DSA.
DC        A(STATIC CSECT)       Constant - address of static internal CSECT filled in
                                by linkage editor.
L         3,16(0,15)            Set up R3 as static base.
L         1,76(0,13)            Set R1 to old NAB (start of new DSA).
L         0,12(0,15)            Place length required for new DSA in R0.
ALR       0,1                   Add old NAB (in R1) and length required for DSA (in
                                R0).
CL        0,12(0,12)            Compare with EOS in TCA.
BC        13,48(0,15)           Branch around library call if new DSA fits segment.
L         15,116(0,12)          Load address of stack overflow routine (IBMBPGRC) from
                                TCA.
BALR      14,15                 Branch to overflow routine.
L         15,16(0,13)           Restore R15 to previous value.  (May have been changed
                                by library call)
ST        0,76(0,1)             Store new NAB in new DSA.
ST        13,4(0,1)             Place backchain in new DSA.
MVC       72(4,1),72(13)        Move address of LWS from old DSA to new DSA.
LR        13,1                  Point register 13 at new DSA.
ST        5,88(0,13)            Set up static backchain.
MVI       87(13),X'91'
MVI       86(13),X'91'
MVI       85(13),X'C0'          Set up enable cells - see chapter 7.
MVI       87(13),X'C0'
MVI       0(13),X'80'           Set up housekeeping flags - see appendix A.
MVI       1(13),X'00'
LA        4,176(0,13)           Set up base for temporaries.

Other code as required          Other tasks may be carried out at this point.  (Such
                                as, initialization of variables with the initial
                                attribute, acquiring a VDA for adjustable variables,
                                and setting up certain error-handling fields.)
BALR      2,0                   Set R2 as program base.
```

Figure 2.8. Typical prologue code

Handling Flow of Control

In PL/I, five types of statement can result in non-consecutive flow of control. These statements are:

- CALL statements
- END statements
- RETURN statements
- Function references
- GOTO statements

The first four of these are concerned with the block structure of the PL/I program and involve passing control from one block to another. GOTO statements can result in branches to code that is either in the current block, or in any other active block.

Consecutive flow of control also ceases when an error or program interrupt occurs. The methods used to handle error and PL/I condition situations are described in chapter 7, "Error Handling."

ACTIVATING AND TERMINATING BLOCKS

CALL, END, and RETURN statements, and function references all result in the activation or termination of blocks. The block structure of PL/I, as explained in chapter 1, is implemented by means of a hierarchy of DSAs.

Each block (begin block, procedure block, or on-unit block) executes on its own program base that is set up at the end of the prologue code for each block. This base is marked in the object code listing with:

```
*   PROCEDURE BASE
```

In the PL/I optimizing compiler, blocks are always called by means of a BALR instruction on registers 14 and 15. Within the prologue code, the registers are stored in the DSA of the calling block, and a new DSA is set up to hold the automatic variables of the new block plus a certain

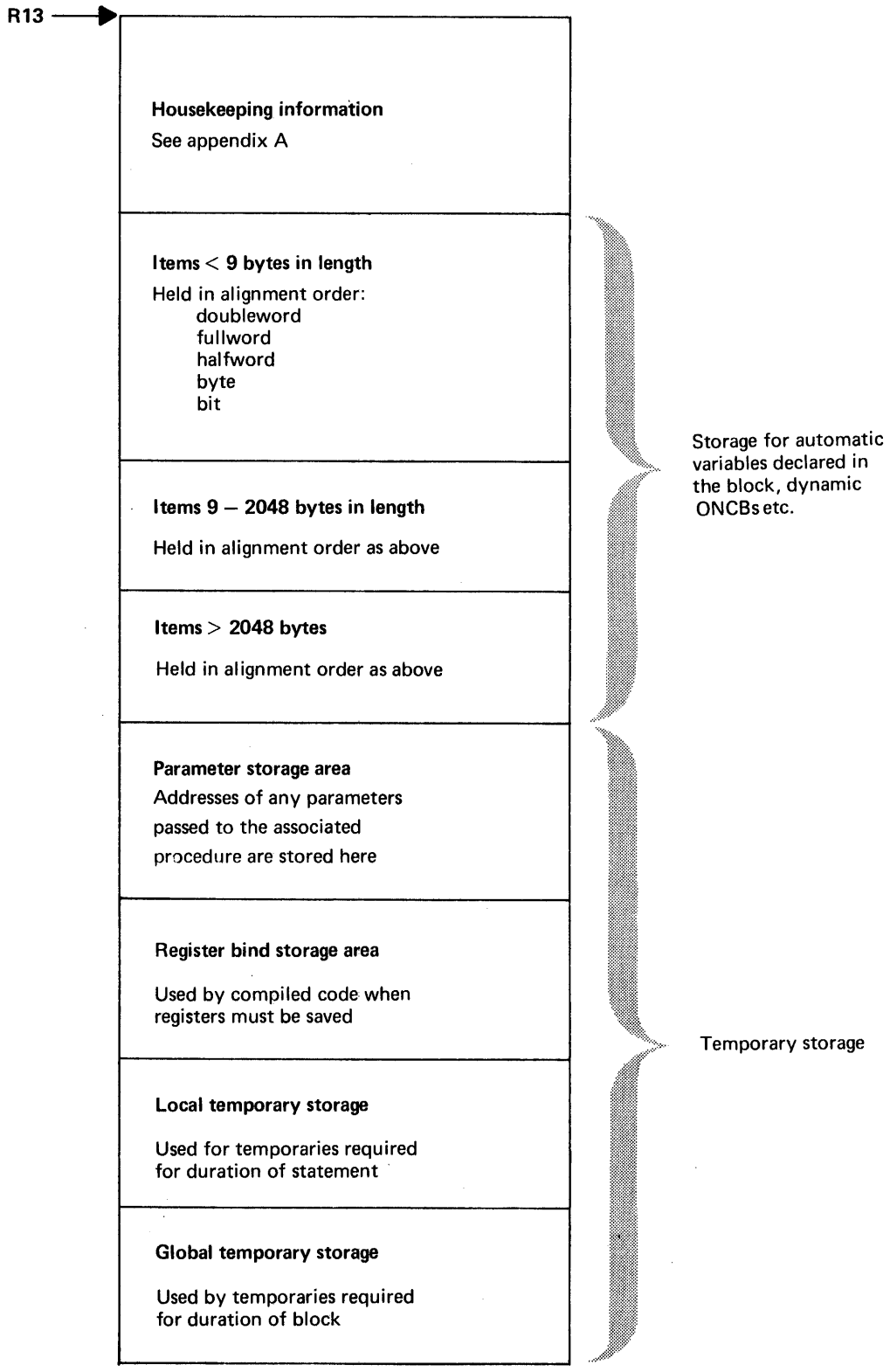


Figure 2.9. Contents of typical compiled code DSA

amount of environmental information such as the enablement or disablement of certain conditions.

When a block is terminated, the registers of the calling block are restored, and a branch is made on register 14. This immediately returns control to the instruction after the BALR issued in the preceding block. The DSA of the called block is automatically discarded because all fields in the DSA, including the pointer to the next available byte of free storage, were addressed from register 13. Because register 13 has been altered, the values that apply to the calling block automatically become current when the calling block's registers are restored.

PROLOGUE AND EPILOGUE CODE

Except for certain single statement on-units, every PL/I begin block or procedure block has a prologue and an epilogue. The prologue prepares the environment for the associated block and acquires storage for automatic variables, compiler-generated temporaries, and workspace. The epilogue frees the storage acquired for the block, restores the registers of the caller, and returns control to the caller.

Prologue

The prologue appears on the object-program listing between REAL ENTRY and PROCEDURE BASE or BLOCK BASE. Every prologue has to acquire a dynamic save area (DSA) for the new block. (The DSA is a register save area concatenated with housekeeping information, plus storage for automatic variables and temporaries.) Other jobs that may be done in the prologue code are:

- Initialization of automatic variables that have the INITIAL attribute.
- Initialization of pointers and locators that have the INITIAL attribute.
- Movement of parameter addresses passed to the procedure to the correct location.
- Acquisition of storage for adjustable variables.
- Initialization of certain items for argument lists.
- Setting-up certain interrupt-handling information such as ONCBs and enable cells. (See chapter 7.)

An example of prologue code is shown in figure 2.8.

After saving the registers, the prologue tests to see if there is enough room for the DSA in the current segment of storage. This is done by adding the length of the new DSA, calculated at compile time, to the address of the next available byte. If the result is greater than the end-of-segment pointer (EOS) placed in the TCA during initialization, the library overflow routine (IBMBPGR) is called to try to to acquire a further segment from the free-area chain. If space for the DSA is available, the next-available-byte pointer (NAB) is updated to point at the first 8-byte boundary beyond the end of the new DSA. The remaining instructions set up housekeeping fields and point registers at various standard fields, including register 13 to the start of the new DSA, and register 4 to the start of storage for temporaries. The final BALR instruction establishes register 2 as the program base register.

Two backchains are set up. The dynamic backchain, which points to the DSA of the calling or preceding block, and the static backchain, which points to the DSA of the statically encompassing block. For the main procedure, the dynamic backchain points to the dummy DSA, and the static backchain is set to zero. The address of the statically encompassing block is passed in register 5.

Static backchains are used in tracing the scope of names and the enablement of PL/I conditions.

For PL/I procedures with COBOL or FORTRAN in the OPTIONS option, the prologue is considerably different. See chapter 13, "Interlanguage Communication."

The format of the DSA is shown in figure 2.9; full details are shown in appendix A.

Epilogue

Epilogue code consists of the instructions generated for END or RETURN statements. These instructions restore the registers to the values that were held when the current block was called. The register values are those stored in the previous DSA. Typical epilogue code is shown in figure 2.10.

The completion of a main procedure results in the raising of the FINISH condition, and this may result in the

execution of an on-unit. Consequently, the address of the current DSA and the address of the current statement must be retained (the DSA is needed to search for the on-unit; the address of the current statement is needed if a SNAP trace is requested in the FINISH on-unit). Epilogue code for a main procedure therefore takes a different form to that generated for a subroutine.

Epilogue code for main procedure		
LR	0,13	Save current DSA address
L	13,4(0,13)	Chainback
L	14,12(0,13)	Pick up value of R14
LM	2,12,28(13)	Restore registers 2 through 12
BALR	1,14	Branch to initialization routine retaining current address in R1
Epilogue code for subroutine or begin block		
L	13,4(0,13)	Chainback
LM	14,12,12(13)	Restore registers of preceding block
BR	14	Return

Figure 2.10. Epilogue code

CALL Statements

CALL statements are executed by picking up the address of the block to be called from static storage. A BALR instruction is then carried out on registers 14 and 15. If arguments are being passed to the called procedure, an argument list is set up in temporary storage, the first bit of the last argument is set to '1', and register 1 is pointed at the argument list.

Typical code would be:

```
00031A  18 50            LR   5,13
            Load static backchain address

00031C  58 F0  3 020     L    15,A...X
            Pick up address of procedure X

000320  05 FF            BALR 14,15
            Branch to procedure
```

Function References

Function references are compiled in exactly the same way as CALL statements. If the function returns a value, an extra field is placed as the last argument in the list. The returned value is placed in this field when the function is completed. Typical code would be:

```
0001FE  41 90  6 0B4     LA   9,B

000202  50 90  3 0BC     ST   9,188(0,3)

000206  41 90  6 0B0     LA   9,A

00020A  50 90  3 0C0     ST   9,192(0,3)
            Set up parameter list

00020E  18 56            LR   5,6
            Load static backchain address

000210  41 10  3 0BC     LA   1,188(0,3)
            Point register 1 at parameter
            list

000214  58 F0  3 008     L    15,A...DOUBLE
            Place address of function
            (DOUBLE) in R15

000218  05 EF            BALR 14,15
            Branch to function
```

END Statement

END statements result basically in restoring the registers of the calling block and branching to the value held in register 14 of that block.

Code compiled for an END statement of an internal block takes the following form:

```
000402  58 D0  D 004     L    13,4(0,13)
            Pick up DSA backchain

000406  98 EC  D 00C     LM   14,12,12(13)
            Restore registers

00040A  07 FE            BR   14
            Branch to procedure
```

For main procedures, certain further actions have to be taken. Because the end of a main procedure raises the FINISH condition, it is necessary to save the current value of register 13 so that the error handler may search the DSA chain for a FINISH on-unit. As it is possible to request a SNAP trace in a FINISH on-unit, it is also necessary to save the address of the END statement. For this reason, the branch is made with a BALR instruction rather than a branch instruction as used

for internal blocks. Typical code would be:

```
00188C  18 0D              LR   0,13
            Save current DSA address in R0

00188E  58 D0  D 004  L    13,4(0,13)
            Pick up DSA backchain

001892  58 E0  D 00C  L    14,12(0,13)
            Restore register 14

001896  98 2C  D 01C       LM   2,12,28(13)
            Restore registers 2 through 12

00189A  05 1E              BALR 1,14
            Branch to initialization
            routine saving branch address
            in register 1
```

RETURN Statement

RETURN statements are executed in a similar way to END statements, but result in the termination of a procedure rather than a block. Consequently, before the restoration of the registers, a chainback must be made to the correct DSA. A chainback is made through any begin blocks. The depth of nesting can be determined during compilation, so the backchain can be loaded the required number of times before the branch is made.

Typical code would be:

```
0003F0  58 D0  D 004       L    13,4(0,13)
            Pick up DSA backchain

0003F4  98 EC  D 00C       LM   14,12,12(13)
            Restore registers

0003F8  07 FE              BR   14
            Branch to procedure
```

Note: If the procedure in which the RETURN statement occurs is a main procedure, the code will take the form compiled for an END statement for an external procedure (see above.)

GOTO STATEMENTS

The implications of a GOTO statement depend on whether the label branched to is within the block or external to it. If the label is outside the block, the branch implies that one or more blocks must be terminated. If the label in the GOTO statement is a label variable, it is not always possible to determine during compilation whether the label will be in the same block as the GOTO statement. Consequently, interpretive code is used for label variables.

For GOTO statements to a label constant within the block, the compiler produces a straightforward branch instruction. For GOTO statements that may pass control to another block, compiled code calls the interpretive code in the TCA.

Interpretive code to handle a GOTO out of block is held in the TCA. To implement a GOTO that will or may transfer control out of the block, compiled code branches to code in the TCA. The code in the TCA checks to see whether it is one of a small number of special cases, and, if it is, calls a library routine -- IBMBPGO. In other circumstances, the GOTO code in the TCA handles the branch and any block termination involved.

GOTO within a Block

The optimizing compiler produces code that assumes that the registers retained across the execution of a labeled statement will be 2, 3, 12, and 13. These are the program base, the static base, the temporary base, the address of the TCA, and the address of the current DSA. All other register values may be different when control passes through the labeled statement on different occasions.

The enablement of conditions may differ in the GOTO statement and in the labeled statement. Within a block, the enablement status may be varied only for the duration of a single statement. The GOTO therefore resets the block enablement status before the branch is taken. If the labeled statement has a different enablement status from the block, it will be automatically reset in the labeled statement.

As explained in chapter 7, "Error and Condition Handling," the enablement of conditions is recorded by enable cells. Two sets are used: the block enable cells retain the enablement situation at the start of the block, which can consequently be restored at any time; the current enable cells hold the enablement situation that is current, which, as explained earlier, may differ from that at the start of the block.

A GOTO within block normally takes the form of a simple branch instruction plus any alteration of the enablement bits that may be necessary to reset the enablement situation to that at the start of the block. Typical code would be:

```
000F1A  47 F0  2 0C8       B    INPUT
```

```
        Branch to correct address in
        compiled code (label name is
        "INPUT")
```

The optimizing compiler attempts to retain the same block base for all branches within a block. However, this is not always possible and, if the code for the block is longer than 4096 bytes, it may be necessary to set up a new base when a GOTO statement is executed. As all labels are stored with both their address and their base this presents no problem. The address of the label and the value of its base form the value of the label constant. The value of the base is placed in register 2, and a branch is made to the label address.

When a GOTO to a label within the block is made, there is no need to reset registers 3, 4, 12, or 13 as these are not altered within a block. When OPTIMIZE(TIME) is specified an attempt is made to retain register values across labels.

Labeled statements within a block have an effect on optimization in that, apart from the bases and block addresses mentioned above, values cannot be retained in registers beyond a labeled statement.

GOTO out of Block

GOTO statements that transfer control from a block have to overcome the problems described above, plus problems of block termination.

For a GOTO out of block or to a label variable, compiled code makes a call to the GOTO code in the TCA, which is held at offset 128 (decimal). The GOTO code receives, through registers 14 and 15, either the contents of the label variable or the equivalent information for a label constant, namely the address at which the label constant is held, and the address of the DSA of the block in which the label appears.

The GOTO code restores registers 3 and 4 from the DSA passed to it, loads register 2 from the second word of the label constant, and loads register 13 from register 15. It then branches to the appropriate point in code which is picked up from the address of the label constant, passed in register 14.

The enablement situation at the start of the block has to be be restored, and this is done by setting the current enable cells in the DSA to the value of the block enable cells. If the current enable cells indicate that CHECK is enabled, a search is made for a qualified CHECK ONCB, so that the enable cells may be set to the start-of-block situation in this ONCB.

In a similar manner, it may be necessary to restore the NAB value to that at the start of the block. This will be necessary if the statement that invoked the block acquired a VDA. The start-of-block NAB value is retained in the DSA and is known as the end-of-prologue NAB. If a VDA has been acquired, the fact is flagged in the flag byte of the DSA, and the GOTO places the end-of-prologue NAB value in the current NAB field.

Such action is never required within a block, as VDAs are only acquired for the duration of one statement and are never used for GOTO statements. Typical code would be:

```
     GOTO label-constant (out of block)

000226  18 E6              LR   15,6
             Place address of DSA in R15

000228  41 E0  3 088       LA   14,136(0,3)
             Place address of label
             constant in R14

00022C  47 F0  C 080       B    128(0,12)
             Branch to GOTO code in TCA
```

GOTO Label Variable

GOTO label variable statements are treated in different ways depending on whether optimization has been specified.

For NOOPTIMIZE, they are all treated as GOTO out of block; for OPTIMIZE (TIME), a check is made to determine whether they could be out-of-block branches. The check is made by testing a label list, which is a list of the label constants to which the label variable may be assigned. If the programmer has supplied a label list, it is used. Otherwise, a list is generated containing all the label constants that are assigned to label variables. If a branch to any of the labels in the list could result in a GOTO out-of-block, all GOTO statements referring to the label variable are treated as GOTO out-of-block situations. Typical code would be:

```
     GOTO label-variable

0000D0  98 EF  D 0A8       LM   14,15,168(13)
             Load R14 and R15 with label
             variable

0000D4  47 F0  0 080       B    128(0,12)
             Branch to GOTO code in TCA
```

Errors when Using Label Variables

Although it is invalid PL/I, it is possible for a GOTO statement using a label variable to result in transfer of control to an inactive block. The optimizing compiler has no method of checking such errors, whose consequences are unpredictable. Such errors can occur because a label variable is not reset when the block containing the label constant to which it refers is terminated. When an attempt is made to GOTO a label variable, the address of the DSA is passed in register 14. The GOTO code assumes this address to be the address of an active DSA, and acts accordingly. Three possibilities arise:

1. The original DSA will not have been overwritten, and the program will execute.

2. The original DSA will have been overwritten with the DSA of another block. The results are then unpredictable, as the code branched to will be accessing an incorrectly mapped DSA.

3. The original DSA will have been overwritten with other information. Again, the results are unpredictable, but may result in an interrupt in the error handler because the backchaining will not be correctly set up.

It should be noted that, because of the method used to allocate DSAs, the chances of one DSA starting at the same address as a previous DSA are high.

GOTO-only On-Units

As explained in chapter 7, certain on-units are not executed as separate program blocks. Instead, the required action is taken under the control of the error handler. On-units containing only a GOTO statement (GOTO-only on-units) are handled in this way.

The error handler accesses on-units through control blocks known as ON control blocks (ONCBs). The ONCB for a GOTO-only on-unit is specially flagged, and the last word of the ONCB is initialized to hold an offset. At this offset in the DSA of the block containing the on-unit, the address of the label information is held. For a label variable, the offset contains the address of the label variable; for a label constant, the offset contains the address of a label temporary that is initialized to the value of the label constant. The initialization is done during the execution of the prologue of the block that contains the on-unit.

The error handler loads the information in the label variable or the label temporary into registers 14 and 15, and calls the GOTO code in the TCA.

Interpretive GOTO routines

If the test in the GOTO code in the TCA reveals that an abnormal situation exists, the interpretive GOTO routine is called. This routine is a subroutine of the program initialization routine.

Two abnormal cases can arise:

GOTO out of SORT exit routine

GOTO from an event I/O on-unit (certain cases only)

When either of these situations could occur a flag is set in the TCA. Sort exits are also flagged in the DSA of the procedure involved.

The SORT exit DSA requires special action because the GOTO will involve the termination of SORT if it transfers control to another block.

The GOTO during an event I/O on unit can cause the termination of a number of WAIT statements. This involves removing information about these statements from the various chains that are set up during event I/O.

These two situations are explained further under the headings "SORT/MERGE" and "WAIT" in chapter 11.

If CHECK enablement has to be changed during an abnormal GOTO, the library routine IBMBPGO is called by the interpretive GOTO routine. To handle the situation. IBMBPGO is described in the licensed publication OS/360 PL/I Resident Library Program Logic.

Argument and Parameter Lists

In PL/I usage, a parameter list is a list of the items a program expects to be passed; an argument list is a list of the items that are passed by the calling routine.

Between PL/I routines, addresses are

always passed rather than the arguments themselves. For strings, structures, arrays, and areas, the addresses of locators are passed rather than the addresses of the arguments themselves. The format of locators and the reasons for their use are given in chapter 4.

When arguments are passed to routines whose entry points are declared with the ASSEMBLER, COBOL, or FORTRAN attributes, the address of the data itself must be passed. The method used is described in chapter 13 "Interlanguage Communication".

Arguments are passed in an argument list addressed by register 1. Normally the list is set up in static storage. The addresses are loaded into consecutive registers and placed in the list by an STM instruction. If the procedure is reentrant or recursive, the list is moved into the temporary storage area in the DSA before the call is made.

The addresses passed in the argument list are moved into the parameter storage area, which is held at the head of temporary storage and is addressed by register 4. (See figure 2.8.) Parameters are then accessed by picking up the addresses from this area.

Dummy arguments, when they are required, are set up by the calling program. Consequently, the called program can treat all arguments in the same manner.

LIBRARY CALLS

Library calls are a feature of every object program. All library calls that appear in the object-program listing are to modules in the resident library. Transient library routines are called by bootstrap routines which are held in the resident library.

The number of library calls used depends on the source program and the level of optimization specified. For OPTIMIZE (TIME), the minimum number of library calls will be made. If NOOPTIMIZE is specified, library calls will be made where this will speed compilation. The standard default is NOOPTIMIZE.

Figure 2.11 shows examples of sequences used for calling library modules. The majority of library calls can easily be recognized by the appearance in the listing of the letters "IBMB" followed by four letters specifying the module name and entry point. To call a module, its address is loaded into register 15, and a BALR instruction is carried out on registers 14 and 15.

```
LA    1,40(0,4)        Point R1 at argument
                       list
LA    14,VO..U(11)     Store address of
                       argument in register
LA    15,DED..VO..     Store address of
      U(11)            argument in register
STM   14,15,0(1)       Load into argument
                       list
L     15,A..IBMBSLOA   Pick up address of
                       routine from static
                       internal control
                       section and place in
                       R15
BALR  14,15            Branch and link to
                       routine
```

Example 1. Call to library routine that has been link-edited and whose address is held in the static internal control section. The arguments passed are addressed by register 1.

```
L     15,116(0,12)     Load address of
                       routine held in TCA
BALR  14,15            Branch and link to
                       routine
```

Example 2. Call to library routine whose address is held in the TCA

Figure 2.11. Examples of library calling sequences

The fifth letter of the entry point name is mnemonic, indicating the type of module that is being called. Figure 2.12 gives the meaning of the mnemonics. Full details of the library modules are given in the program product publications OS PL/I Transient Library: Program Logic and OS PL/I Resident Library: Program Logic.

A further discussion of library module naming conventions is given chapter 3.

Setting-Up Argument Lists

Before a call is made to a library module, an argument list must normally be set up. This is done in one of several ways, depending on the library module. The majority of library calls require the method shown in figure 2.11, example 1. This consists of loading the list into sequential registers starting at register 14, and then using a store-multiple instruction to place the arguments into an area of static storage, whose address is then loaded into register 1. Argument

lists are set up as far as possible during compilation and, where necessary, completed during execution.

IBMBA---	Array handling
IBMBB---	String handling
IBMBC---	Conversion
IBMBE---	Error handling
IBMBI---	Interlanguage communication
IBMBJ---	Date/time/delay/wait
IBMBK---	Dump/sort/checkpoint/restart
IBMBM---	Mathematical
IBMBO---	Open/close
IBMBR---	Record I/O
IBMBS---	Stream I/O
IBMBT---	Completion pseudovariable routine

Figure 2.12. Mnemonic letters in library module entry-point names

Offset from start of TCA (Register 12)		Name of module entry point	Use
Decimal	Hex		
72	48	IBMBPGRD	Stack overflow routine to get VDA
84	54	IBMBEFL	FLOW module
108	6C	IBMBPGRA	Get non-LIFO dynamic storage
112	70	IBMBPGRB	Free non-LIFO dynamic storage
116	74	IBMBPGRC	Stack overflow routine for prologue
120	78	IBMBERRB	Error handler software interrupt
264	108	IBMBJWTA	WAIT module
268	10C	IBMBTOCA	Completion pseudovariable routine
272	110	IBMBTOCB	Event variable assignment routine

Figure 2.13. Offsets where addresses of library modules are held in the TCA

Addressing the Subroutine

As can be seen in example 1 of figure 2.11, library addresses are generally held in static storage and addressed as an offset from register 3. However, the addresses of certain library routines are held in the TCA or the TCA appendage and addressed from register 12. They are addressed either directly or indirectly as shown in example 2 of figure 2.11. The names of these routines do not appear on the listing; however, they can be identified by their offset from the start of the TCA (see figure 2.13).

DO-LOOPS

Where possible, do-loops are carried out by means of a BXLE instruction, because this is more efficient than using a simple BCT instruction. BXLE do-loops can be used where the control variable cannot be altered except at the head of the loop, and where it is not subsequently accessed after the completion of the loop. BXLE do-loops cannot be used for the outer of a number of nested do-loops. For outer loops, other branch instructions are used. Code for a number of typical do-loops is shown below. Note that the code will differ according to the content of the loop.

Source program

```
     DO I = 1 to 10;

       DO J = 1 to 10;
            .
            .
            .
            .
            .
          END;

       END;
```

Object program

1. Code for outer do-loop

```
      LH        5,596(0,3)   Pick up 1 from
                             constants pool

      STH       5,I          Place 1 in I
CL.1  EQU       *
        .
        .
        .
        .
        .
```

```
        LH      5,I
        AH      5,596(0,3)      Increment and
        STH     5,I             store in I
        C       5,598(0,3)      Compare I and
                                constant 10 in
                                static storage
        BNH     CL.1
```

2. Code for inner do-loop

```
        LH      5,596(0,3)      Place 1 in
                                first operand
        LH      10,596(0,3)     Place 1 in
                                second operand
        LH      11,598(0,3)     Place 10 in
                                comparand
CL.2    EQU     *
         .
         .
         .
         .
         .
        BXLE    5,10,CL.2       Increment,
                                test, and
                                branch if
                                necessary.
```

Compiler-Generated Subroutines

The compiler uses internal subroutines to carry out certain functions. These have the advantage over library modules, because they can be tailored for the most common case. When special cases arise, the library routines are called. Compiler-generated subroutines have the further advantage that they are internal to compiled code and consequently need not follow the standard operating system calling sequence.

Compiler-generated subroutines are used for the following purposes.

IELCGIA	Stream I/O input - provides address of source of next edit-directed data or format item
IELCGIB	Stream I/O input - housekeeping after transmission of data item
IELCGOA	Stream I/O output - provides address of target of next edit-directed data or format item
IELCGOB	Stream I/O output - updates FCB, counts data item, and frees VDA if one was used
IELCGOC	Stream I/O - processes X format items
IELCGMV	Move long (registers 6,7,8,9)
IELCGCL	Compare long (registers 1,6,7,8,9)
IELCGCB	Compare long bits
IELCGON	Dynamic ONCB chaining
IELCGRV	Revert VDA chaining
IELCGBB	Test for '1' bits
IELCGBO	Test for '0' bits

Compiler-generated subroutines are held in separate control sections and are printed at the head of the object-program listing when they are used in a program.

Optimization and its Effects

Optimization is the attempt to produce the most efficient possible object program. The OS PL/I Optimizing Compiler adopts a threefold approach:

1. It attempts to compile each statement in the most efficient manner.
2. It modifies the resulting code for each block, in an attempt to make it more efficient (for example, by maintaining values in registers and by using common control blocks for similar items).
3. It examines the source program to discover whether statement flow can be reorganized to produce a more efficient program (for example, by moving code out of loops).

The effect of specifying the compiler option OPTIMIZE (TIME) is that the compiler loads and calls the optimization phases, and executes optimization code in other phases. The optimization phases are described in the publication OS PL/I Optimizing Compiler: Program Logic.

When NOOPTIMIZE is specified, the optimization phases are not called; no attempt is made to study the flow of the program, and the examination of compiled code for possible improvements is not undertaken on a global basis. More library calls will generally be made if NOOPTIMIZE is specified.

EXAMPLES OF OPTIMIZED CODE

A number of the more noticeable effects of optimization are shown below. These show code sequences which may prove difficult to understand without knowledge of the objectives of optimization. Where possible, the examples of code given are expansions of the examples shown in the language reference manual for this compiler. The examples do nct cover all optimization

carried out by the compiler.

Elimination of Common Expressions

Elimination of common expressions is handled by avoiding multiple calculations of the same expression, the value being retained either in temporary storage or in a register. In the examples shown below, the common expression is "B+C". In the first example, the value is held in a register. In the second, it is held in temporary storage, because the value to which it is first assigned is altered. In certain circumstances, the code could be compiled to move the value from the variable to which it was originally assigned to the second variable.

Example 1: Value held in register

Source program

```
2     A=B+C;

3     IF X<Y THEN X=Y;
4     D=B+C;
```

Object program

```
* STATEMENT NUMBER  2
000062  78 00 D 0A4       LE    0,B
000066  7A 00 D 0A8       AE    0,C
00006A  70 00 D 0A0       STE   0,A

* STATEMENT NUMBER  3
00006E  78 60 D 0AC       LE    6,X
000072  79 60 D 0B0       CE    6,Y
000076  47 B0 2 020       BNL   CL.2
00007A  78 60 D 0B0       LE    6,Y
00007E  70 60 D 0AC       STE   6,X
000082              CL.2  EQU   *

* STATEMENT NUMBER  4

* CALCULATION OF COMMONED EXPRESSION
FOLLOWS
000082  70 00 D 0B4       STE   0,D
```

Example 2: Value held in temporary storage

Source program

```
2          A=B+C;
3          IF X<Y THEN A=6;
4          D=B+C;
```

Note: A may be altered before subsequent use of expression.

Object program

```
* STATEMENT NUMBER  2
000062  78 00 D 0A4       LE    0,B
000066  7A 00 D 0A8       AE    0,C
00006A  70 00 4 028       STE   0,40(0,4)
00006E  70 00 D 0A0       STE   0,A

* STATEMENT NUMBER  3
000072  78 60 D 0AC       LE    6,X
000076  79 60 D 0B0       CE    6,Y
00007A  47 B0 2 024       BNL   CL.2
00007E  78 00 3 010       LE    7,16(0,3)
000082  70 00 D 0A0       STE   7,A
000086              CL.2  EQU   *

* STATEMENT NUMBER  4

* CALCULATION OF COMMONED EXPRESSION
FOLLOWS
000086  78 20 4 028       LE    2,40(0,4)
00008A  70 20 D 0B4       STE   2,D
```

Movement of Expressions out of Loops

When expressions cannot be altered inside a section of code that may be executed a number of times, the expression is moved out of the loop to a position where it will be executed only once, regardless of the number of times that the loop is executed. The process is known as movement of invariant expressions. The most obvious example is in do-loops. However, the compiler analyzes the source program for other types of loop and also moves code from these.

Example 1 shows code moved from a do-loop. Example 2 shows code moved from a loop that has been detected by the compiler. It should be noted that code moved out of loops frequently involves conversion and is not obvious in the source program.

Example 1: Do-loop

Source program

```
2          DO I=1 TO N;
3          J=3;
4          END;
```

Object program

```
* STATEMENT NUMBER  2
000066  48 E0 D 0A2       LH    14,N
00006A  18 BE             LR    11,14
00006C  48 A0 3 014       LH    10,20(0,3)
000070  48 50 3 014       LH    5,20(0,3)
000074  19 5B             CR    5,11
000076  47 20 2 024       BH    CL.3
```

```
Source program

   2          DCL C(10) FLOAT DECIMAL (6);
   3          DCL B(10) FLOAT DECIMAL (6);
   4           DO I=1 TO 10
   5            C(I)=B(I);
   6            END;
Object program

 * STATEMENT NUMBER  4
000066  48 60 3 010       LH    6,16(0,3)          Pick up 1 from static
00006A  40 60 0 0A0       STH   6,I                Place in I

 * INITIALIZATION CODE FOR OPTIMIZED LOOP FOLLOWS

 * CODE MOVED FROM STATEMENT NUMBER 5
00006E  48 E0 3 012       LH    14,18(0,3)         Load "4" into R14 from static
000072  48 90 3 014       LH    9,20(0,3)          Load "40" into R9 from static
000076  18 B9             LR    11,9               Load "40" into R11 for BXLE
000078  48 A0 3 012       LH    10,18(0,3)         Load "4" into R10
00007C  18 5E             LR    5,14               Load "4" into R5

 * CONTINUATION OF STATEMENT NUMBER  4
00007E              CL.2  EQU   *

 * STATEMENT NUMBER  5
00007E  78 45 D 0A4       LE    4,VO.,B(5)         Pick up VO..B+R5
000082  70 45 D 0CC       STE   4,VO..C(5)         Place in VO..C+R5

 * STATEMENT NUMBER  6
000086  87 5A 2 018       BXLE  5,10,CL.2          Increment R5 by 4, test for end of
                                                   loop, and branch or continue
```

Figure 2.14. Modification of do-loop control variable

```
 * INITIALIZATION CODE FOR OPTIMIZED LOOP
 FOLLOWS

 * CODE MOVED FROM STATEMENT NUMBER 3
 00007A  48 70 3 016       LH    7,22(0,3)
 00007E  40 70 D 0A4       STH   7,J

 * CONTINUATION OF STATEMENT NUMBER   2
 000082              CL.2  EQU   *

 * STATEMENT NUMBER  4
 000082  87 5A 2 01C       BXLE  5,10,CL.2
 000086  40 50 D 0A0       STH   5,I
 00008A              CL.3  EQU   *
```

Example 2: Compiler-detected loop

Source program

```
   2          L: IF X>Y THEN GOTO BED;
                 /*LOOP BEGINS*/
   3             J=I-N;
   4          X=X+J;
   5       GO TO L;    /*LOOP ENDS*/
   6       BED:   A=X;
```

Object program

```
 * INITIALIZATION CODE FOR OPTIMIZED LOOP
 FOLLOWS
 * CODE MOVED FROM STATEMENT NUMBER 3
 000066  48 E0 D 0AE       LH    14,I
 00006A  4B E0 D 0B0       SH    14,N
 00006E  50 E0 4 028       ST    14,40(0,4)

 * STATEMENT NUMBER  2

 * STATEMENT LABEL  L
 000072  78 00 D 0A0       LE    0,X
 000076  79 00 D 0A4       CE    0,Y
 00007A  47 20 2 042       BH    BED

 * STATEMENT NUMBER  3

 * CALCULATION OF COMMONED EXPRESSION
 FOLLOWS
 00007E  58 60 4 028       L     6,40(0,4)
 000082  40 60 D 0AC       STH   6,J

 * STATEMENT NUMBER  4

 * END OF COMMON CODE
 000086  50 60 4 030       ST    6,48(0,4)
 00008A  48 60 3 020       LH    6,32(0,3)
```

```
Source program

   2          IF (A=D)|(C=D) THEN
                 X=Y+Z;
Object program

 * STATEMENT NUMBER  2
 000062  78 00 D 0A0       LE    0,A                Pick up A
 000066  79 00 D 0A4       CE    0,D                Compare A and D
 00006A  47 80 2 018       BE    CL.3               Branch if equal
 00006E  78 40 D 0A8       LE    4,C                Pick up C
 000072  79 40 D 0A4       CE    4,D                Compare C and D
 000076  47 70 2 024       BNE   CL.2               Branch if not equal
 00007A              CL.3  EQU   *
 00007A  78 60 D 0B0       LE    6,Y
 00007E  7A 60 D 0B4       AE    6,Z                  X=Y+Z
 000082  70 60 D 0AC       STE   6,X
 000086              CL.2  EQU   *
```

Figure 2.15. Branching around redundant expressions

```
Source program

   2          X=123;   /*COMMONED ITEM*/
   3           Y=123*Z;
   4          V=V**123;
   5          A=123;   /*COMMONED ITEM*/
Object program

 * STATEMENT NUMBER  2
 000066  78 00 3 020       LE    0,32(0,3)              /*COMMONED ITEM*/
 00006A  70 00 D 0A0       STE   0,X

 * STATEMENT NUMBER  3
 00006E  78 20 D 0A8       LE    2,Z
 000072  6C 20 3 018       MD    2,24(0,3)
 00007A  70 20 D 0A4       STE   2,Y

 * STATEMENT NUMBER  4
 00007E  41 10 D 0AC       LA    1,V
 000082  41 50 3 024       LA    5,36(0,3)
 000086  41 60 D 0AC       LA    6,V
 00008A  58 F0 3 00C       L     15,A..IBMBMXSA
 00008E  05 EF             BALR  14,15

 * STATEMENT NUMBER  5
 0000B8  78 20 3 020       LE    2,32(0,3)
 0000BC  70 20 D 0B0       STE   2,A                    /*COMMONED ITEM*/
```

Figure 2.16. Use of common constants

```
 00008E  40 60 4 030       STH   6,48(0,4)        * STATEMENT LABEL  BED
 000092  97 80 4 032       XI    50(4),X'80'      0000A8  70 00 D 0A8       STE  0,A
 000096  78 60 4 030       LE    6,48(0,4)
 00009A  7B 60 3 020       SE    6,32(0,3)
 00009E  3A 60             AER   6,0
 0000A0  70 60 D 0A0       STE   6,X
```

Elimination of Unreachable Statements

```
 * STATEMENT NUMBER  5
 0000A4  47 F0 2 00C       B     L

 * STATEMENT NUMBER  6
```

If the source program contains statements that can never be executed because they are unconditionally branched around, these

statements will be ignored by the compiler.

In the example below, the statements between 5 and 8 can never be reached. Consequently, no code is compiled for these statements, and a compiler diagnostic message is issued to indicate that this is the case.

Example

Source program

```
5  GOTO LABEL;
6    IF A<B THEN
                IF B<C THEN
                            IF A<X THEN
                              B=B*C;
7    ELSE C=B*C;
8 LABEL:  X=X+1;
```

Object program

```
  * STATEMENT NUMBER  5
  00008A  47 F0 2 028       B     LABEL

  * STATEMENT NUMBER  8

  * STATEMENT LABEL  LABEL
  00008E  78 60 D 0AC       LE    6,X
  000092  7A 60 3 018       AE    6,24
                                  (0,3)
  000096  70 60 D 0AC       STE   6,X
```

Compiler message reads:

"6,6,6,7 STATEMENT MAY NEVER BE EXECUTED. STATEMENTS IGNORED."

Simplification of Expressions

Certain expressions are simplified for speedier execution. For example, multiplication is simplified to addition, as in the following example.

Example: Multiplication into addition

Source statement

```
2    X=3*B
```

Object program

```
  * STATEMENT NUMBER  2
  000065E 78 20 D 0AC       LE    2,B
  000062  3A 22             AER   2,2
  000064  7A 20 D 0AC       AE    2,B
  000068  70 20 D 0A8       STE   2,X
```

Modification of DO-loop Control Variables

When the do-loop control variable is used for accessing array elements, it is frequently modified to simplify addressing of the array elements.

If, as in the example in figure 2.14, the elements of the array are four bytes long, it simplifies addressing to increment the loop control variable by 4 rather than by 1. When this is done, the increment becomes the distance between the start of successive array elements. Provided that the original value of the loop control variable is the same as that of the first bound of the array, the loop control variable in turn becomes the offset of the element from the virtual origin of the array.

If the loop control variable is altered, this means that the increment and final value must also be altered. Thus the loop in the example instead of being incremented from 1 to 10 by 1, is incremented from 4 to 40 by 4. Note that the value of the loop control variable is set at the start of the loop but is not incremented. If the value of the loop variable is required after the loop has been executed, this type of optimization cannot take place.

In the example, the control variable is held in register 5 using a BXLE instruction. The array elements are addressed by using register 5 as the offset from the virtual origins of arrays C and B. As register 5 starts the loop with the value of 4 and is incremented by 4 for each iteration of the loop, this gives the correct address. Both arrays begin 4 bytes from their virtual origins, and each array element is 4 bytes long.

Branching around Redundant Expressions

If a series of tests are to be made and action taken if any of the tests proves positive, the compiler takes the requisite action as soon as the first positive test is found.

In the example in figure 2.15, a test is first made to see if A=D. If so, the value of Y+Z is assigned to X without a further test being made to see if C=D. Note that the last test is for inequality, so that if the variables are equal, control will continue with the code that assigns the value to X.

Rationalization of Program Branches

When the length of a program is greater than 4096 bytes and, consequently, it cannot be addressed from one base register, an attempt is made to update the base at the most efficient point, so that there will be as few changes of program base as possible during execution. The aim is to avoid any program branches which move from the scope of one base register to the scope of another.

The program base register is register 2, and this is updated when necessary. As register 2 is required for in-line record I/O and TRT instructions, the program base is saved and restored after such use.

Use of Common Constants and Control Blocks

Constants and control information used more than once are generated only once in static storage. Thus for the statements X=768, Y=768, the constant value 768 will be picked up from the same address in both cases. Similarly, compiler-generated control information, such as DEDs and descriptors (see chapter 4), are generated only once if a number of variables require identical control information.

The process of avoiding duplication is known as commoning. It should be noted that constants may not be commoned if they are not used in the same way. In the example in figure 2.16, constant '123' is stored in a different form for assignment, multiplication, and exponentiation.

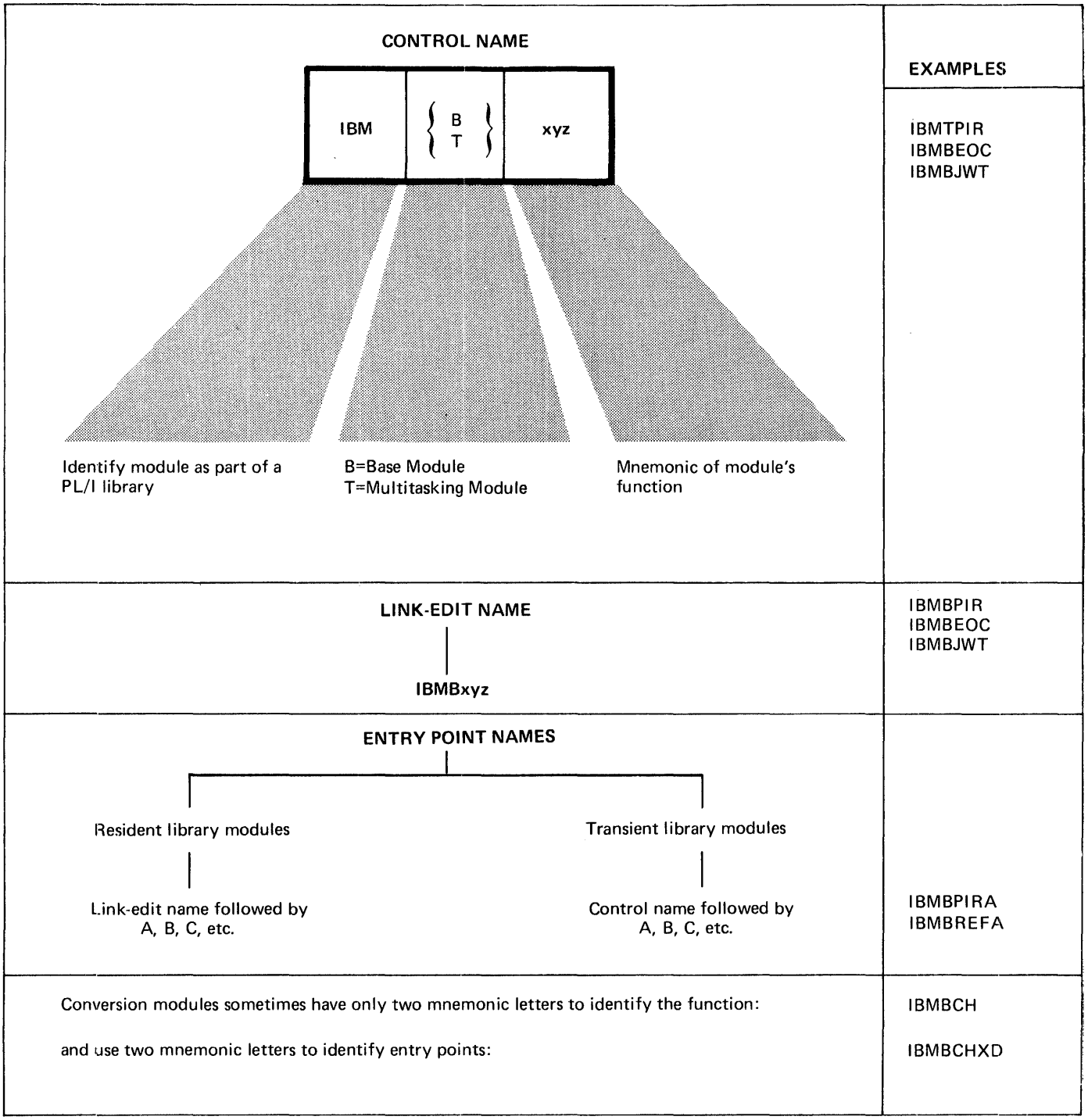


Figure 3.1. Library module naming conventions

Chapter 3: The PL/I Libraries

This chapter explains the use of libraries by the OS PL/I Optimizing Compiler. The topics covered are: when and why library routines are called, why there is both a transient library and a resident library, naming conventions, and two implementation topics that cover all library modules: the use of library workspace and the use of weak external references. Also covered are the multitasking and shared libraries.

The OS PL/I Optimizing Compiler is designed to be used in conjunction with the OS PL/I Resident Library and the OS PL/I Transient Library. These libraries consist of sets of standard subroutines that are used for the majority of interfaces with the system and for those jobs that can be most efficiently done by the use of interpretive subroutines. The main areas where library modules are used are: input/output, error handling, storage management, conversions, mathematical functions, and various string- and array-handling operations.

Use of library routines simplifies compilation by enabling the compiler to set up an argument list and generate a call to a subroutine, rather than compile the complete code. However, library subroutines are less efficient than compiled code, since they must be generalized routines, whereas compiled code can be specially tailored to the particular program being executed. Furthermore, a library call involves the overhead of saving and restoring registers, and may require the setting-up of various additional control blocks to describe the data (see chapter 4). For these reasons, programs that are optimized for time use as few library calls as possible.

The majority of interfaces between compiled code and the operating system are implemented via library routines. This is done mainly for reasons of implementation convenience, as such interfaces are in this way localized and minimized.

Resident and Transient Libraries

The OS PL/I subroutine library is divided into two separate program products: the OS PL/I Resident Library (Program Number 5734-LM4) and the OS PL/I Transient Library (Program Number 5734-LM5). Resident library modules are link-edited with the executable program phase. Transient library modules are loaded into dynamic storage when they are required; when they are no longer needed, the storage is freed and may be overwritten. Resident library routines have the advantage of speed; transient library routines have the advantage of saving space. By using both types of library, it is possible to produce more efficient programs.

Routines in the transient library are: input/output transmitters, open and close modules, error message modules, the storage management routines and PLIDUMP routines. All other library routines are held in the resident library, including a number of bootstrap routines that load and call transient routines.

The OS PL/I libraries reside on three direct-access data sets. The resident library is on SYS1.PLIBASE and SYS1.PLITASK. The transient library resides on SYS1.LINKLIB.

The internal logic of individual library modules is described in the publications OS PL/I Resident Library: Program Logic and OS PL/I Transient Library: Program Logic. However, in such cases as I/O, error handling, and conversion, where compiled code and a hierarchy of library modules are used in implementing certain features of PL/I, the overall logic is described in this publication. Similarly, an overall explanation of storage management and interlanguage communication is given in this publication.

Naming Conventions

Most PL/I library modules have names of seven letters, the first three letters being IBM. This identifies the module as belonging to one of the PL/I libraries. The remaining letters indicate which particular library the module was written for, and the use of the module.

Each resident library module has two names, the control name (which uniquely identifies the module) and the link-edit name (which appears in the linkage editor map and the object-program listing). The majority of the modules in the OS resident library have a control name with the fourth letter B, for example IBMBOCL. This module has a link-edit name of IBMBOCLA. Some

modules, however, have a fourth letter T in their control name, indicating that they are used only in a multitasking environment. The link-edit names of these modules nevertheless have a fourth letter B. An example of this is the multitasking priority-alteration routine IBMTTPRA. The link-edit name for this module is IBMBTPRA. (See figure 3.1.)

The result of this arrangement is that a number of library modules can share the same link-edit name. Consequently, the compiler can generate the same code regardless of whether the program is going to operate in a multitasking or non-multitasking environment.

Entry point names are given additional letters alphabetically. The primary entry point (of resident library modules) is normally the link-edit name. Other entry points are named "B", "C", etc. For example, the primary entry point of the module with control name IBMBOCL is IBMBOCLA and the secondary entry point is IBMBOCLB.

The naming convention for conversion modules is slightly different. Arithmetic conversion modules have entry points indicated by a two-letter mnemonic code.

The Multitasking Library

The resident library is held on two data sets: SYS1.PLIBASE and SYS1.PLITASK. SYS1.PLIBASE holds all modules that are needed to execute non-multitasking programs. SYS1.PLITASK holds the multitasking versions of all modules that differ for multitasking and non-multitasking environments.

As explained above, both multitasking and non-multitasking modules have the same link-edit names for their entry points. Multitasking modules have a fourth letter T; non-multitasking modules have a fourth letter B, in their control names.

The use of the same link-edit name permits the compiler to generate the same code for library calls, regardless of whether the program is multitasking or non-multitasking. For multitasking programs, the data set SYS1.PLITASK must precede SYS1.PLIBASE in input to the linkage editor. In this way, the multitasking modules will be link-edited and the program will run in a multitasking environment. Further details of this arrangement are given in chapter 14.

Library Workspace

DSAs (dynamic storage areas) for certain library routines are not acquired in the same way as they are for source program subroutines. Instead of the storage being acquired from the LIFO stack, space is allocated, in the program management area, for two pre-formatted DSAs. These DSAs are known as levels of library workspace. Their format can be seen in figure 3.2. Library workspace (LWS), provides a fast method for library routines to obtain DSAs. All the library routines have to do is to address the DSA and set the chainback field. There is no need to test to see if there is enough space for the DSA, as the space is already allocated. The NAB pointer does not have to be reset, because the next available byte is not changed.

The PL/I libraries have been designed so that no more than two library modules require library workspace at any one time. This does not mean, however that no more than two library modules are ever active at any one time. Where more than two modules can be active, one or more of the modules will use a DSA in the LIFO stack. For conversion, a slightly different arrangement is used, whereby a DSA is acquired by the first module in the series and used by subsequent modules. This arrangement is described in chapter 10.

FORMAT OF LIBRARY WORKSPACE

Library workspace is designed so that either level can be treated by the housekeeping routines in the same way as a DSA. Chainback fields to the calling block's save areas are held in the head of library workspace and, where more than one level of library workspace is used, a chainback field is set up to the previous level. Figure 3.2 illustrates the method of chaining employed.

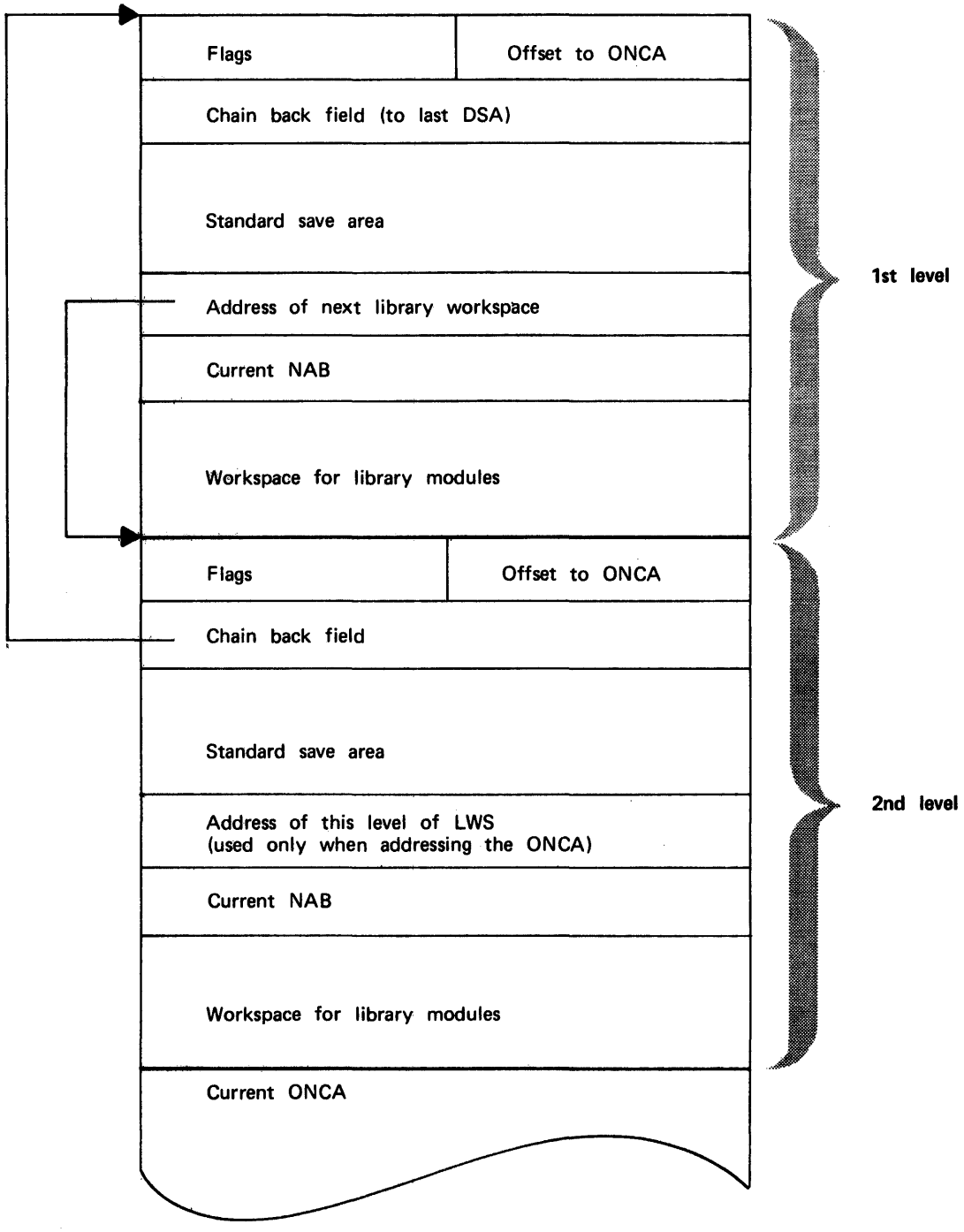


Figure 3.2. Library workspace

ALLOCATION OF LIBRARY WORKSPACE

Library workspace is originally allocated within the program management area by the initialization routine IBMBPII. However, whenever an interrupt occurs and an on-unit is to be entered, a further two levels are allocated. This allows library modules to be called within an on-unit, without overwriting library workspace which may have been in use at the time of interrupt. Library workspace is acquired by a subroutine of IBMBPIR that is addressed from the TCA.

Attached to each allocation of library workspace, including the initial allocation in the program management area, is an ON communications area (ONCA). This is a control block used in error handling to hold condition built-in function values. ONCAs are described fully in chapter 7.

Library Modules and Weak External References

Because of the modular structure of the library, a group of modules is frequently used to carry out some particular task. Conversions, for example, are normally done by using a series of modules, and so are many of the mathematical built-in functions. For this reason, many library modules contain a number of external references to modules which may not be needed in a particular program. An example of this is shown in figure 3.3. To prevent unnecessary modules being link-edited, "weak external references" (WXTRNs) are used. WXTRNs are a special type of external reference designed to cater for this situation.

Those entry points that are called only optionally are coded as WXTRNs. This prevents the linkage editor from loading these modules unless a separate external reference is made to them by the compiler. Thus the executable program phase does not contain modules that it never uses.

Figure 3.3 shows part of a hierarchy of modules with alternative paths through them. When such a hierarchy exists, the actual path to be taken through the modules will be known to the compiler, and external references will be made to all the required modules whose names are coded as WXTRNs. The effect of this is that the linkage editor loads only the required modules.

The Shared Library

The shared library is a PL/I facility that allows an installation to load PL/I resident library modules into the link-pack-area (LPA) so that they are available to all PL/I programs. This reduces space overheads.

The modules to be included in the shared library can be chosen by the installation. They must include the initialization routine, the error handling routine, the open file routine, and all modules addressed from the TCA that are not identical for multitasking and non-multitasking programs. Further details are given in the publication OS PL/I Optimizing Compiler: System Information.

The routines in the shared library are held in two of three link-pack-area modules: IBMBPSM, and either IBMBPSL or its multitasking equivalent IBMTPSL. Each of the link-pack modules contains a number of library routines, and is headed by an addressing control block known as a transfer vector. IBMBPSM contains those modules in the shared library that are common to both multitasking and non-multitasking PL/I environments. IBMBPSL contains the non-multitasking versions of those modules that are not identical in multitasking and non-multitasking PL/I environments. This module has a multitasking counterpart, IBMTPSL, which holds the multitasking versions of such modules.

Two further modules are also involved in handling the shared library. These are the shared library addressing modules IBMBPSR and its multitasking counterpart IBMTPSR (R stands for region). One or other of these modules is link-edited with compiled code and held in the program region: IBMBPSR for non-multitasking programs, or IBMTPSR for multitasking programs. IBMBPSR and its multitasking counterpart hold dummy entry points which duplicate the names of all entry points of modules within the shared library. References to such entry points in compiled code are resolved to the dummy entry points in IBMBPSR or IBMTPSR.

The situation during execution is as shown in figure 3.4. In the link-pack-area are two link-pack modules: IBMBPSM and IBMBPSL (or its multitasking counterpart); these contain all the routines in the shared library. In the program region is the shared library addressing module IBMBPSR (or its multitasking counterpart). All references by compiled code to entry points in the shared library have been resolved by the linkage editor to IBMBPSR (or IBMTPSR).

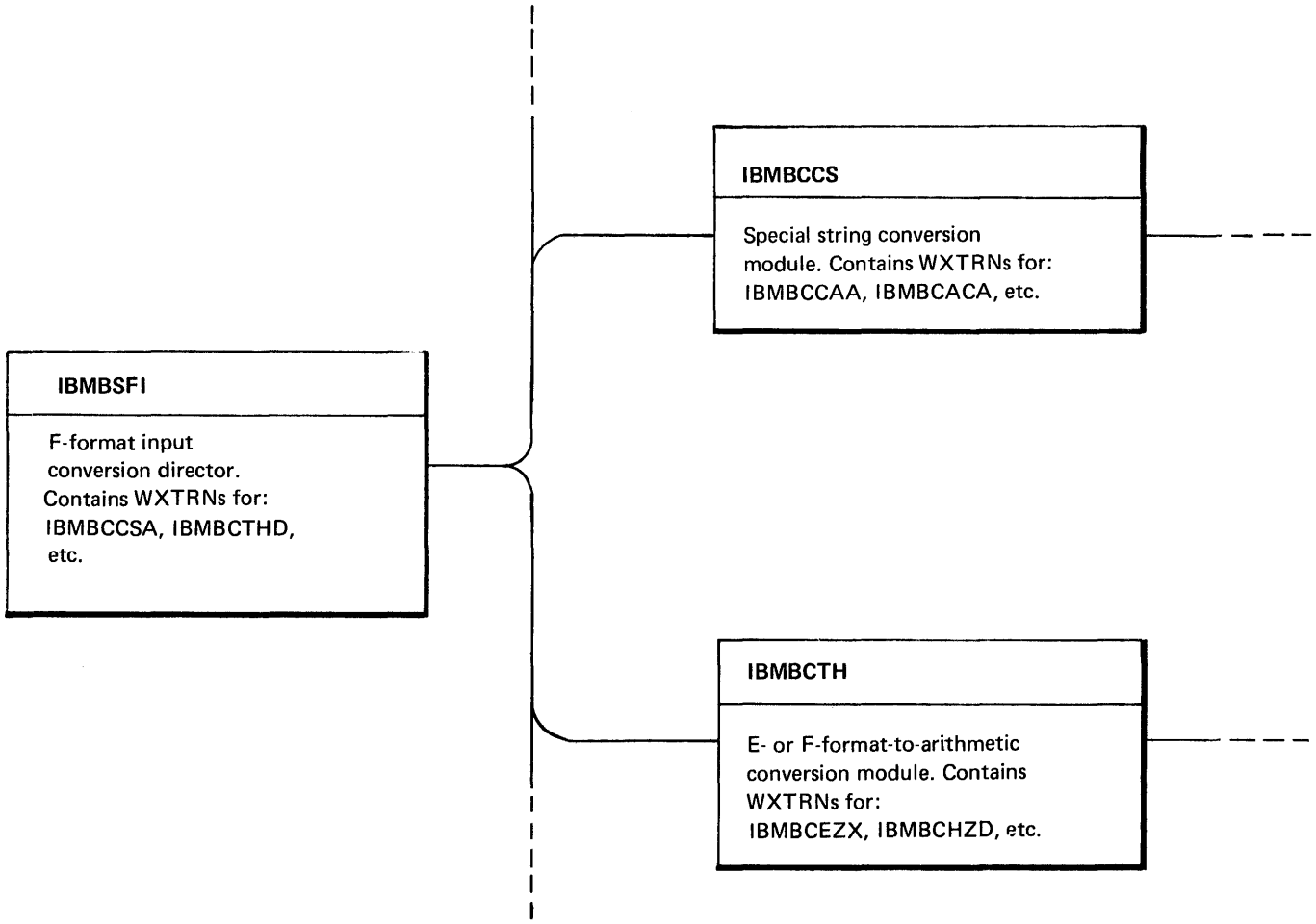


Figure 3.3. Example of use of WXTRNs

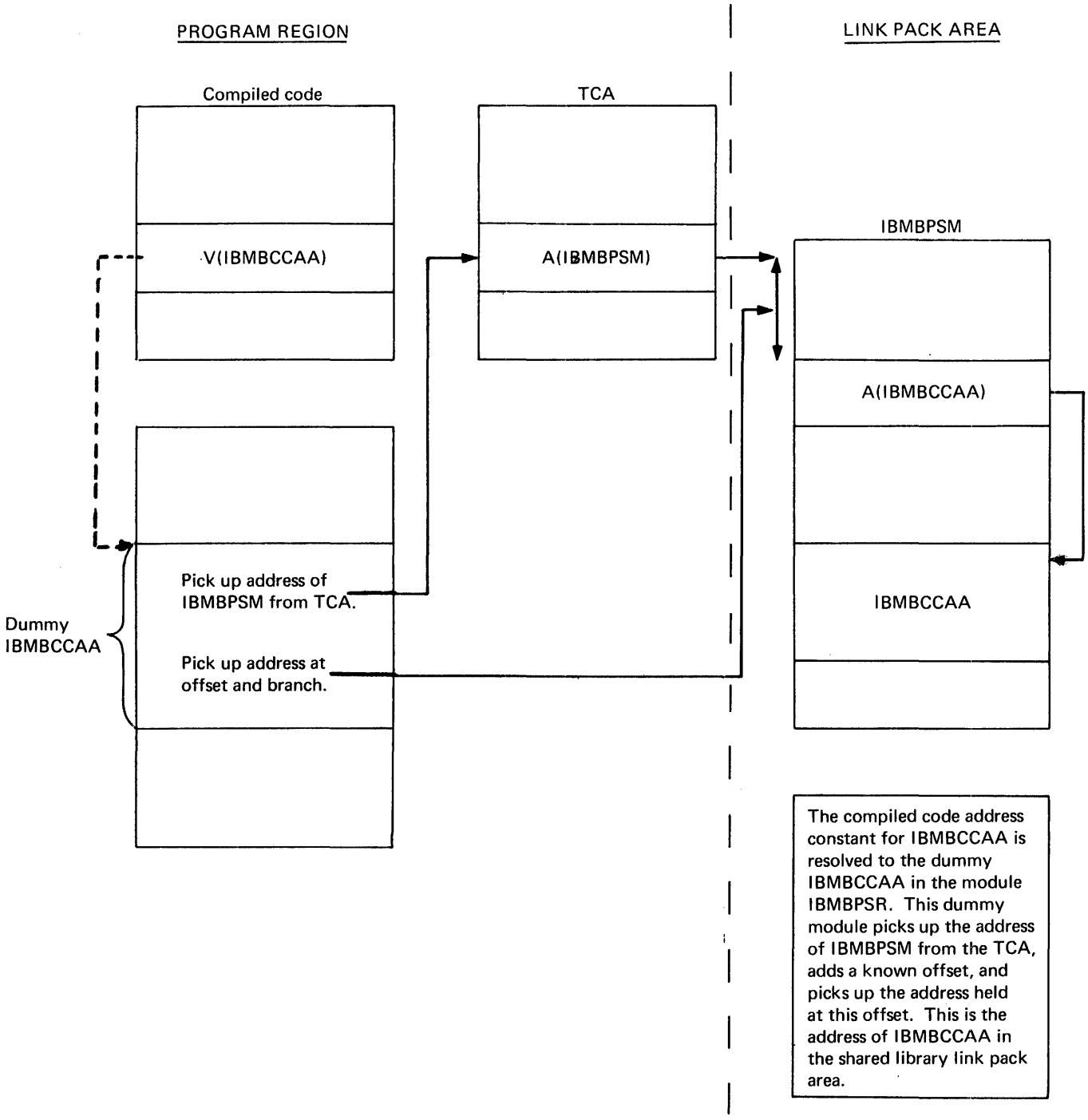


Figure 3.4. The shared library during execution

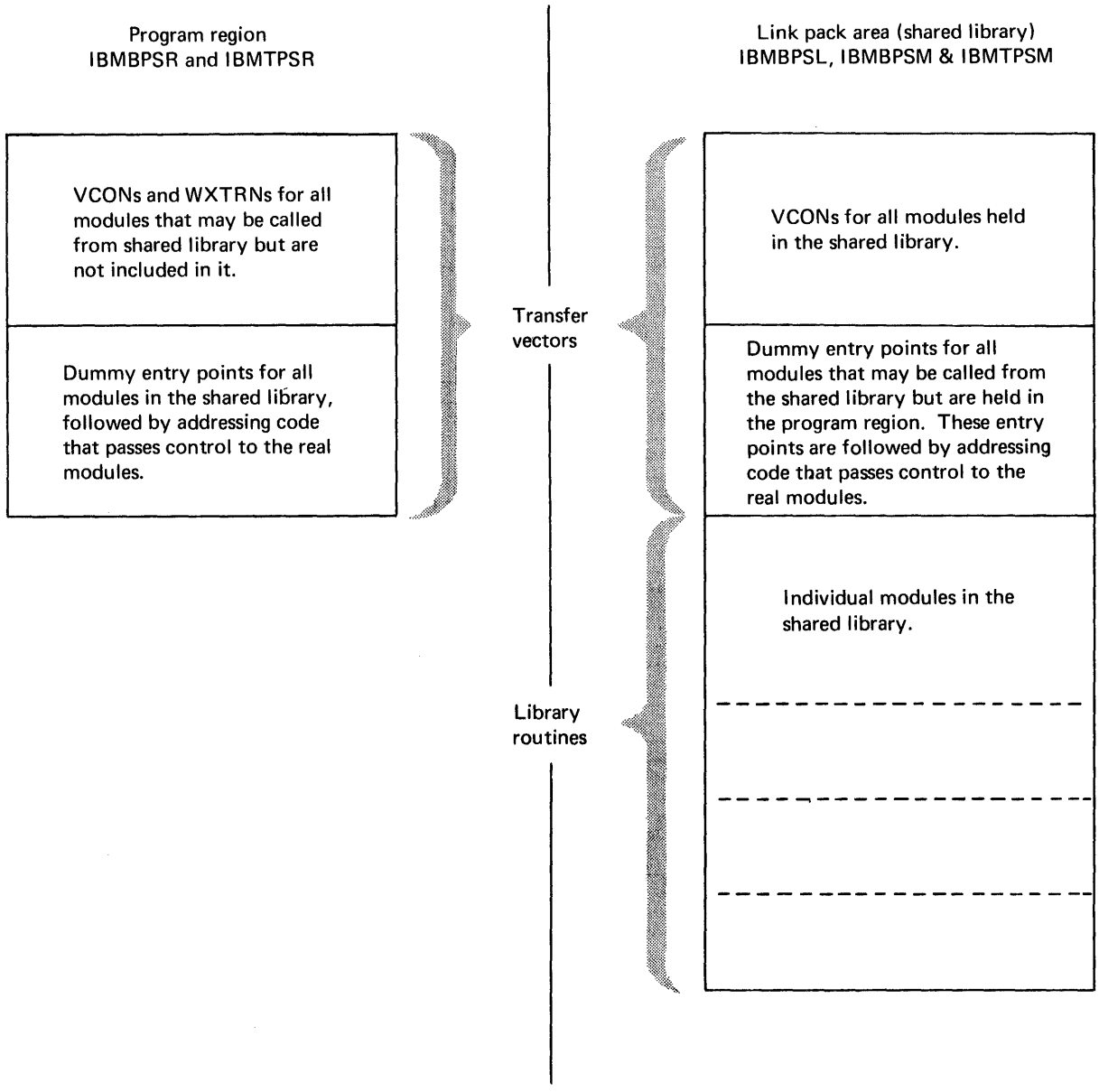


Figure 3.5. The format of shared library modules

Communication between Program Region and Link-Pack-Area

Communication between the link-pack-area and the program region is handled by the transfer vectors that are held at the head of each module. Communication is necessary in both directions. The compiled program will need to call library subroutines that are held within the link-pack modules in the link-pack-area. Similarly, certain of the modules in the link-pack-area may need to call modules that are not included in the shared library.

The link-pack-area modules IBMBPSL and IBMBPSM, are headed by transfer vectors, which are followed by the individual library modules in the shared library. The individual modules and the transfer vector are link-edited to form one module when the shared library is created. The program region module IBMBPSR consists only of a transfer vector. (The format of the shared library modules is shown in figure 3.5.) During program initialization, the addresses of the three modules being used (and consequently the address of the transfer vector) are placed in the TCA.

The transfer vectors contain three types of data:

1. Dummy entry points for all modules that are not held in that area (i.e., the program region transfer vector contains dummies for all entry points that are held in the shared library; the link-pack transfer vector contains entry points for all modules that could be called from the shared library but are not included in it).

2. Code, following the dummy entry points, that passes control from the dummy entry point in one area to the real entry point in another area. The code takes the form:

```
L   15, offset(12)

L   15, xxx(15)

BR 15
```

   where "offset" is the offset to the address of the transfer vector in the TCA, and "xxx" is the offset within the transfer vector to the required address.

3. An ordered list of addresses for all routines that are held in the same area as the vector.

The code (item (2) above) transfers control in the manner shown in figure 3.5.

1. It picks up the address of the relevant transfer vector from the TCA, where it was placed during program initialization.

2. It picks up the address of the module it requires from a known offset from the start of the transfer vector.

3. It branches to the address, thus passing control to the required library routine.

The code does not use any register except register 15. The link register (14) is not altered, and control returns directly from the module to the caller.

Execution when Using the Shared Library

Use of the shared library is specified by the linkage editor statement INCLUDE PLISHRE. PLISHRE is an alias for the program region modules IBMBPSR and IBMTPSR. The appropriate module will therefore be loaded by the linkage editor (IBMBPSR for non-multitasking programs; IBMTPSR for multitasking programs). All compiled code external references to shared library module entry points are then resolved to the dummy entry points in IBMBPSR (or IBMTPSR). Similarly WXTRNs in the program region module are resolved if compiled code issues an EXTRN for the entry point.

Program Initialization

At the start of the program, control is passed to one of the entry points of the initialization routine. This entry point will, in fact, be a dummy entry point in the shared library program region module. Each entry point is followed by code which requests the system to load the shared library link-pack modules. If the modules are already loaded, the system simply returns their addresses. If they are not loaded, it loads them into the link-pack-area, and then returns the addresses.

The addresses of the two link-pack-area modules and of IBMBPSR are added to the parameter list for IBMBPIR. IBMBPIR is then called in the usual shared library manner, that is, via the transfer vector in one of the link-pack modules.

It is the standard action of the initialization routines to load these parameters into the appropriate fields in the TCA. When the shared library is not in use, meaningless information is loaded into

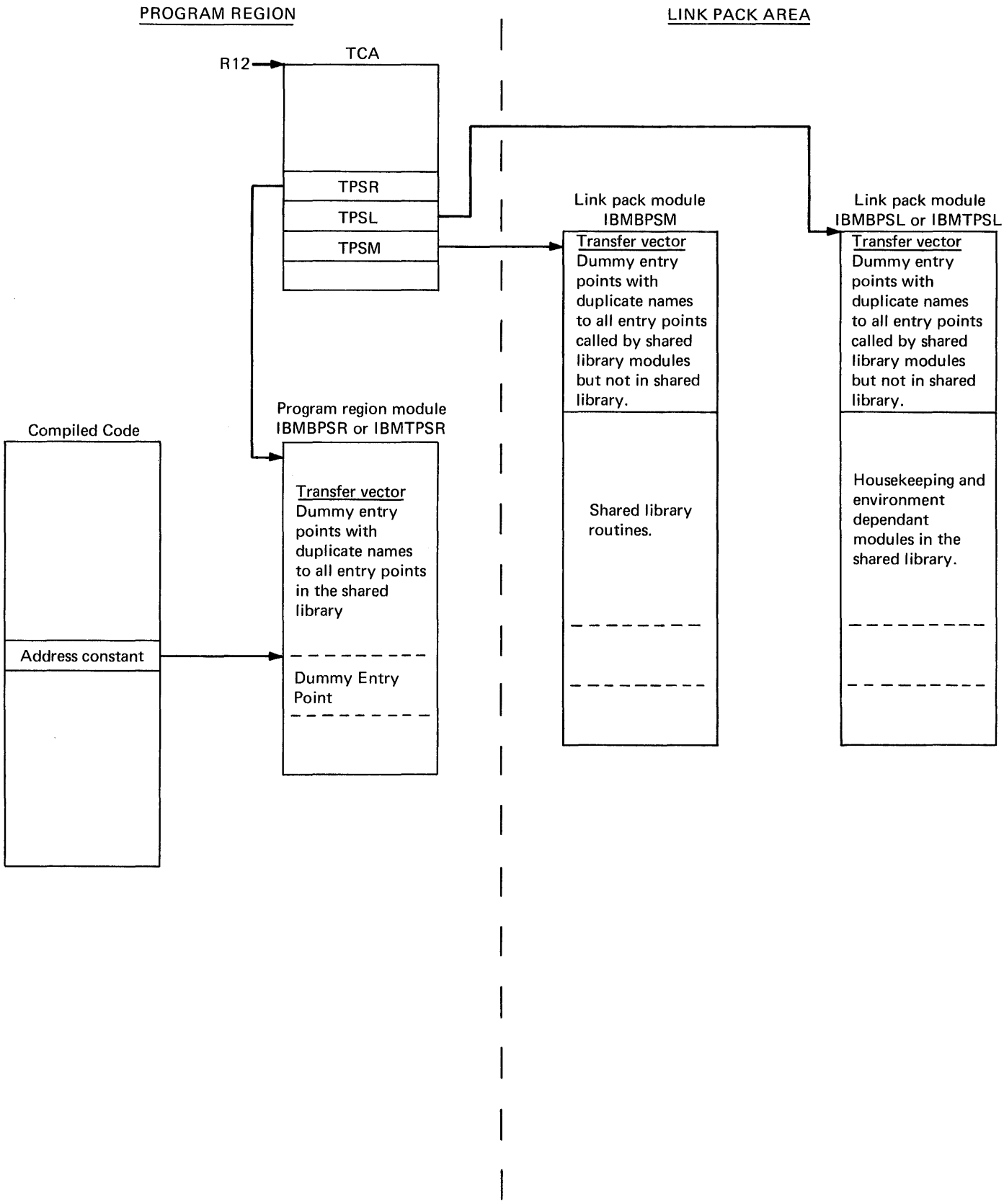


Figure 3.6. Addressing a module in the shared library

these fields. However, as they are only accessed by the shared library modules, this does no harm.

Initializing the Shared Library

The shared library is initialized by the use of special macro instructions, as described in the System Information manual.

All five modules must be created at the same time. During the process, the table of VCONs in the link-pack modules, transfer vectors are generated, and the offsets to these VCONs from the head of the transfer vector are placed in the code following the dummy entry points in the program region modules. A similar process is carried out for addresses in the program region. The VCONs within the link-pack modules are resolved by the linkage editor when the link-pack modules are created. The VCONs within the program region modules are qualified by WXTRNs, and are only resolved if compiled code generates an EXTRN for the entry point. Such EXTRNs are generated when required, as a normal part of the compilation process, regardless of whether the shared library is being used. The VCONs in the program region modules are resolved by the linkage editor when the program is link-edited.

Multitasking Considerations

The shared library has been designed so that multitasking does not affect it. If PLI.TASK is specified before PLI.BASE, the linkage editor statement INCLUDE PLISHRE will result in the module IBMTPSR being loaded and linked in the program region. When control passes to the code following the IBMBPIR entry point in IBMTPSR, a request is made to the system to load the multitasking shared library module IBMTPSM. The program then runs in the usual manner, with the multitasking modules.

An installation can specify a shared library that includes only the multitasking or the non-multitasking modules. However both multitasking non-multitasking versions of the program region module will still be created. The module for the unwanted environment will be a dummy. This prevents problems should an INCLUDE PLISHRE statement be included in a program that is intended to run in the environment with no shared library. If this process was not carried out, such a statement could result in the incorrect environment being initialized.

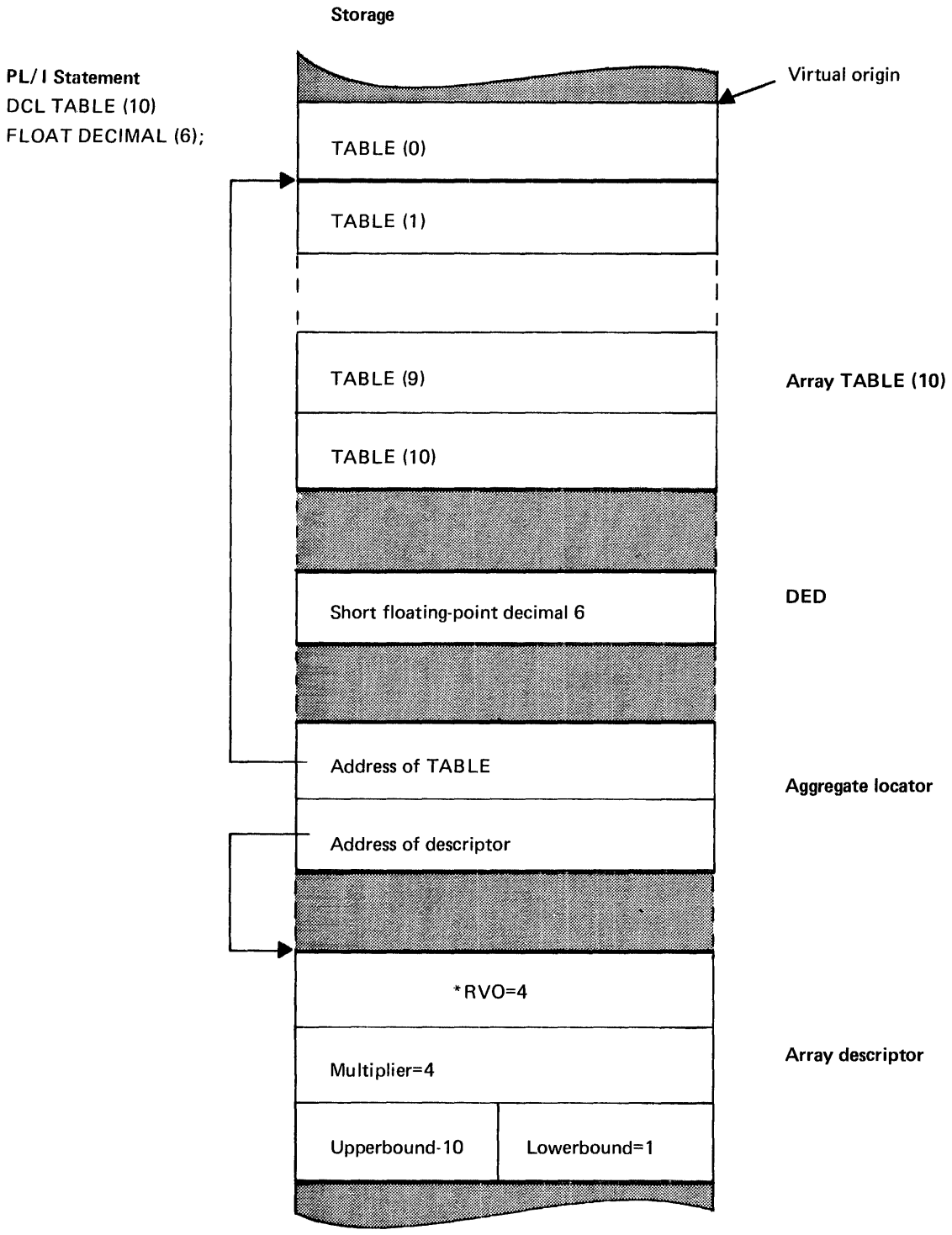


Figure 4.1. Example of descriptor, locator, DED, and storage location of an array

Chapter 4: Communication Between Routines

PL/I allows the programmer the choice of a large number of data attributes. Normally there is no need for explicit attribute information to be retained until execution, because the methods used to handle the data can be resolved during compilation. However, there are certain situations where this cannot be done. For example, adjustable bounds or extents may prevent the data attributes being fully known at compile time, or the data may be being passed to another PL/I procedure or library subroutine. When these situations arise, it is necessary to retain some or all of the data attributes in an explicit form throughout execution.

The names of variables fall into a similar category. Normally, they need not be explicitly known during execution. However, for data-directed input/output and the CHECK condition, the names of the variables need to be known so that they can be associated with the correct values.

When such information must be retained until execution, special control blocks are set up for the purpose. These control blocks are described in this chapter.

The control blocks are:

Descriptors: These hold the extent of the data item (i.e., string lengths, array bounds, and area sizes).

Locators: These hold the address of a data item and, if they are not concatenated with the descriptor, hold its address.

Descriptor Descriptors: These hold the logical structure levels, dimensions, and lengths, of all elements within a structure.

Data Element Descriptors (DEDs): These hold the attributes of a variable required for data manipulation, except for extents, which are held in descriptors.

Symbol Tables: These hold the names of the variables and associate them with the appropriate storage locations during execution.

Symbol Table Vector: This associates symbol tables with the block in which they are known.

Descriptor/Locator: This is a term used to describe the control block consisting of a descriptor concatenated with a locator.

An example of the way in which data is related to its locators, descriptors, and DEDs is given in figure 4.1.

Notes on Terminology

The following terms are used in this chapter.

Virtual origin (VO)	The address where the element of an array whose subscripts are all zero is held or, if such an element does not appear in the array, where it would be held.
Actual origin (AO)	The address of the first item in the array or structure.
Relative virtual origin (RVO)	Actual origin minus virtual origin.
Structure element	A minor or major structure that contains a number of base elements.
Base element	A data element or array within a structure.

DESCRIPTORS AND LOCATORS

Descriptors are generated when adjustable extents are involved, or when an item is to be passed as an argument and the associated parameter is the type that can be declared with an asterisk among its attributes. For example, DCL X CHAR (N); or DCL X CHAR (*); would both result in the generation of a descriptor. In the first case, code for the SUBSTR built-in function would have to be interpretive if STRINGSIZE were enabled. The appropriate library module would be called, and it would make use of the descriptor to discover the length of the string. This length would have been placed in the descriptor by the prologue code of the block in which the string was declared. In the second case, where the length of the string is signified with an asterisk, the program that is passed the string will expect to receive the length of the string in a descriptor.

Name of control block	Conditions under which it is generated	Location (control section)
Data element descriptor (DED)	When conversion or stream I/O library modules are called.	Static internal
Array descriptor	When an array has adjustable bounds or may be passed to a library subroutine or other PL/I routine.	Static internal
Aggregate locator	When structure or array descriptor is generated.	Static internal
Area locator/descriptor	When an area is declared with an adjustable size or may be passed as an argument.	Static internal
String locator/descriptor	When a string is declared with an adjustable length or is passed as an argument.	Static internal
Structure descriptor	When a structure is declared with adjustable elements or is passed as an argument.	Static internal
Aggregate descriptor descriptor	When a structure contains elements declared with adjustable bounds.	Static internal
Symbol table	When an item may appear in data-directed I/O or in a CHECK list	Static internal for internal items. Separate CSECT for external items.
Symbol table vector	When GET DATA or PUT DATA is used without a data list, or when SIGNAL CHECK is used without a data list.	Static internal

Figure 4.2. Descriptors, locators, and symbol tables: when generated, where held

Data items that can be declared with an adjustable value or an asterisk are: string lengths, array bounds, and area sizes. Descriptors are, therefore, needed for strings, arrays, and areas. They are also needed for structures, because structures can contain strings, arrays or areas.

In order to connect the data with its descriptor, a further control block is generated. This is the locator. The locator addresses both the descriptor and the variable. 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 and holds the address of both the variable and the descriptor. Called routines are normally passed the addresses of locators, rather than the addresses of arguments when arguments requiring descriptors are passed.

When the descriptor and locator are not concatenated, it is possible to use the same descriptor for a number of different data items, provided that these items have the same attributes. This process is known as "commoning" and is used to conserve space. Where possible, the compiler commons structure and array descriptors and aggregate descriptor descriptors.

Except for controlled variables, descriptors and locators are always held in the static internal control section, regardless of the attributes of the data that they describe.

For controlled variables, the descriptor and, sometimes, the locator are held immediately before the data. (For details see 'Controlled Variable Control Block' in appendix A).

The following types of descriptor and locator are generated. Figure 4.2 summarizes the conditions under which they are generated and gives their storage

locations. In the main, they are set up during compilation and completed during execution, if necessary.

String Locator/Descriptor

The string locator/descriptor holds the byte address of the string, information on whether or not it is a varying string, and the maximum length of the string. For a bit string, the bit offset from the byte address is held. (See figure 4.3.)

Area Locator/Descriptor

The area locator/descriptor holds the address of the start of the area and the length of the area. (See figure 4.4.)

Aggregate Locator

The aggregate locator holds the address of the start of the array or structure and the address of the array descriptor or structure descriptor. (See figure 4.5.)

Array Descriptor

The array descriptor holds:

1. The relative virtual origin (RVO) of the array. This is the offset of the start of the first element in an array (actual origin) from the virtual origin. The virtual origin (VO) is the point at which element (0) would be held in a one-dimensional array, element (0,0) would be held in a two-dimensional array, etc. In a one-dimensional array, the address of any particular element can be discovered by multiplying together the subscript and the multiplier (see below) and adding the result to the virtual origin of the array. An extension of this method is used for multi-dimensional arrays, the formula being:

$$\text{Address of element } (S_1,S_2,\ldots,S_n) = VO+\sum_{i=1}^{n}(M_i*S_i)$$

where S is the subscript number, and M the multiplier, of the ith dimension, and VO is the virtual origin.

For unaligned bit-string arrays, the virtual origin points to the byte address before the element (0). The bit offset is held in the string descriptor, which is concatenated with the array descriptor.

2. The high and low bounds for the subscripts in each dimension.

3. The multiplier for each dimension. The multiplier is the distance between the start of one element and the start of the next element in the same dimension. For example in the array declared A(2,2), the multiplier for the first dimension is the distance between the start of element A(1,1) and the start of element A(1,2).

When the array is an array of strings or areas the string or area descriptor is concatenated with the end of the array descriptor to provide the necessary additional information. Array descriptors are commoned where possible. That is, one descriptor is used for a number of similar arrays.

Structure Descriptor

The structure descriptor consists of a series of fullwords, giving 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. Where a bit offset is involved, this will be held in the descriptor for the bit string, or in the relative virtual origin if the item is a bit string array.

A structure must be mapped during execution if any of the elements in the structure have adjustable bounds or extents, or if the REFER option is used. Where possible, structure descriptors are commoned. That is, one descriptor is used for a number of similar structures. If a structure or an array of structures contains elements with adjustable extents, the structure descriptor is not set up during compilation. Instead, it is set up during execution from information held in the structure descriptor descriptor. (See below for information on arrays of structures and structures of arrays.)

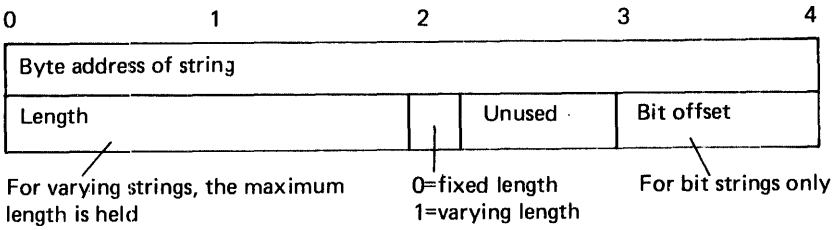


Figure 4.3. String locator/descriptor

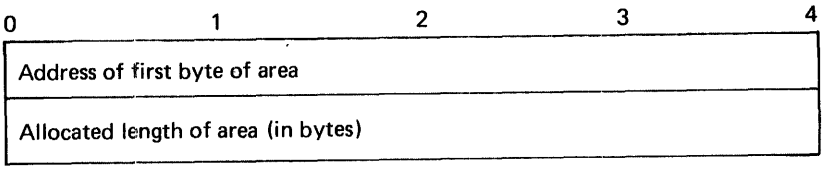


Figure 4.4. Area locator/descriptor

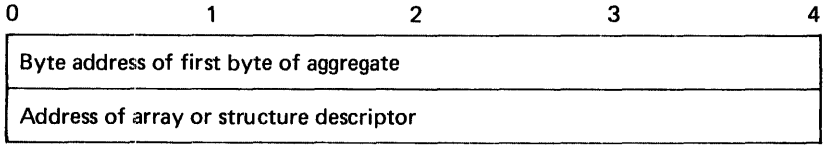
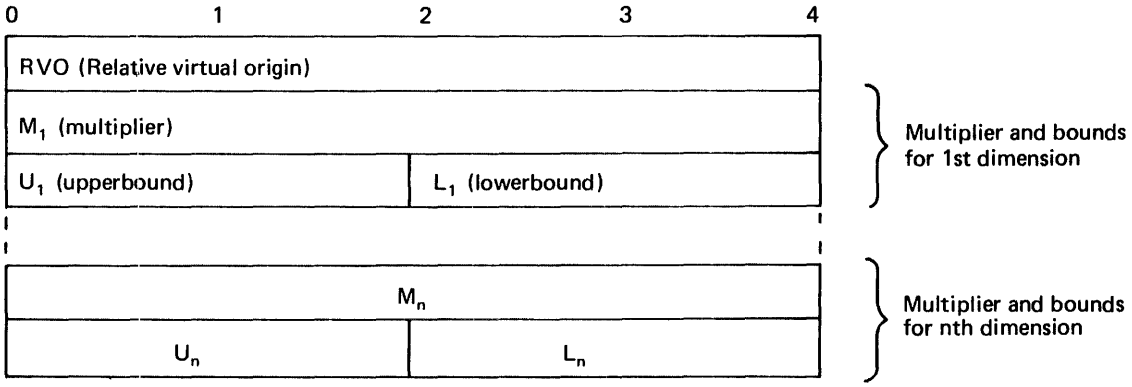


Figure 4.5. Aggregate locator



Notes:
1. For unaligned bit strings, RVO and multiplier are bit values.
2. For strings and areas, the area or string descriptor is concatenated to the end of the array descriptor.

Figure 4.6. Array descriptor

Aggregate Descriptor Descriptor

When a structure cannot be mapped during compilation, more information than is held in the structure descriptor is needed for it to be mapped during execution. This information is held in a control block known as an aggregate descriptor descriptor.

The information held in an aggregate descriptor descriptor is the number of dimensions and logical level of all the structure elements, and the number of dimensions, logical level, and alignment requirements, of all base elements, plus the length of those base elements that do not have their length held in descriptors. (Strings and areas, and arrays of strings and areas, have their lengths in descriptors.) The length held for an array is the length of an array element. The total length of the array can be calculated by using the information in the array descriptor.

The aggregate descriptor descriptor is set up in static internal storage and is set up completely during compilation. The format is shown in figure 4.7. An example showing the method used to map a structure that contains an element with an adjustable extent is shown in figure 4.8.

Where possible, aggregate descriptor descriptors are commoned.

Arrays of Structures and Structures of Arrays

Where necessary, an aggregate locator, a structure descriptor, and an aggregate descriptor descriptor are generated for both arrays of structures and structures of arrays.

The structure descriptor for both an array of structures and a structure of arrays has the same format. The difference is in the values in the fields of the array descriptors within the structure descriptor. Take for example the array of structures AR and the structure of arrays ST, declared below.

Array of Structures	Structure of Arrays
DCL 1 AR(10), 2 B, 2 C;	DCL 1 ST, 2 B(10), 2 C(10);

The structure descriptor for both AR and ST would contain an offset field for both B and C and an array descriptor for both B and C. (See figure 4.9.) However, the values in the descriptors would differ, because the array of structures AR would consist of elements held in the order B,C,B,C, etc., and the elements in the structure of arrays ST would be held in the order:

B,B,B,B,B,B,B,B,B,B,C,C,C,C,C,C,C,C,C,C.

DATA ELEMENT DESCRIPTORS

When data is passed to the PL/I library routines, a complete description of the data is frequently required, and something more than a descriptor is therefore needed. Conversion routines, for example, need to know the complete attributes of the data. To hold such information, data element descriptors (DEDs) are generated. (Control blocks known as DEDs are also used by the compiler. These are compile-time DEDs and have a different format from those that are used during execution. Compile-time DEDs never appear in the executable program.) For stream I/O, DEDs are generated to describe the format of the input or output. These DEDs are known as format element descriptors (FEDs).

DEDs are produced for all types of variable or temporary that are passed to the library for conversion or stream input/output. The length and format of the DED depends on the data type of the item. DEDs are shown in detail in appendix A. An indication of their format is given in figure 4.10.

DEDs are always held in static internal storage. They are used only to pass information to library routines.

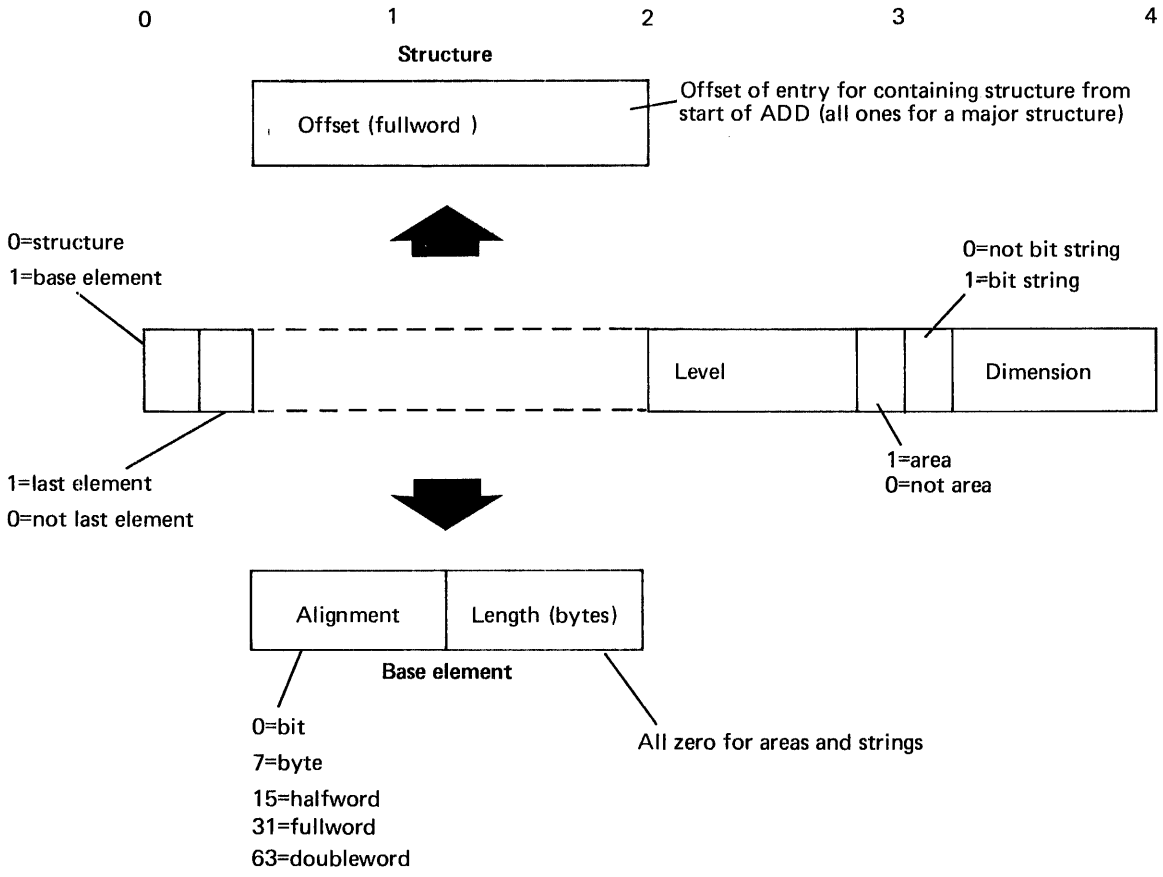
There are five types of DEDs: arithmetic DEDs, arithmetic pictured DEDs, string DEDs, pictured string DEDs, and FEDs.

Arithmetic DEDs: are 4 bytes long.

Arithmetic pictured DEDs: (always decimal) are 8 bytes plus picture specification, which consists of at least one byte for every character in the pictured string. Maximum length for pictured arithmetic DEDs is 264 bytes.

String DEDs: are 4 bytes long.

Pictured string DEDs: (always character string) are six bytes plus the picture specification, which consists of one byte for every character in the picture string. The maximum length for pictured character DEDs is 261 bytes.



There is a fullword entry in the ADD for each structure (major and minor) and each base element.

Figure 4.7. Aggregate descriptor descriptor

DURING COMPILATION

1. Space for structure descriptor allocated in static storage.
2. Aggregate descriptor descriptor allocated, and fields filled in from structure declaration.
3. Aggregate locator allocated, and address of structure descriptor place in second word.

   Code is generated within the prologue of the block in which the structure is declared to call structure mapping routine, IBMBAMM, to acquire a VDA, and to complete the aggregate locator.

DURING EXECUTION

4. Prologue code places value of N(1 byte) in the string descriptor for D in structure descriptor.
5. IBMBAMM is called to map the structure, using the information in the ADD and the SD (which contains the length of element D). D is aligned with E, then B is aligned with DE. (The rules for structure mapping are given in the language reference manual for this compiler.) The results of the mapping are placed in the structure descriptor.
6. IBMBAMM returns the length of the structure to compiled code, which acquires a VDA for the structure and places the address of the structure in the aggregate locator.

THE RESULT

Every member of the structure can be addressed by means of the address in the aggregate locator and the offsets within the structure descriptor. When bit offsets are involved, they are contained within the appropriate descriptor in the structure descriptor.

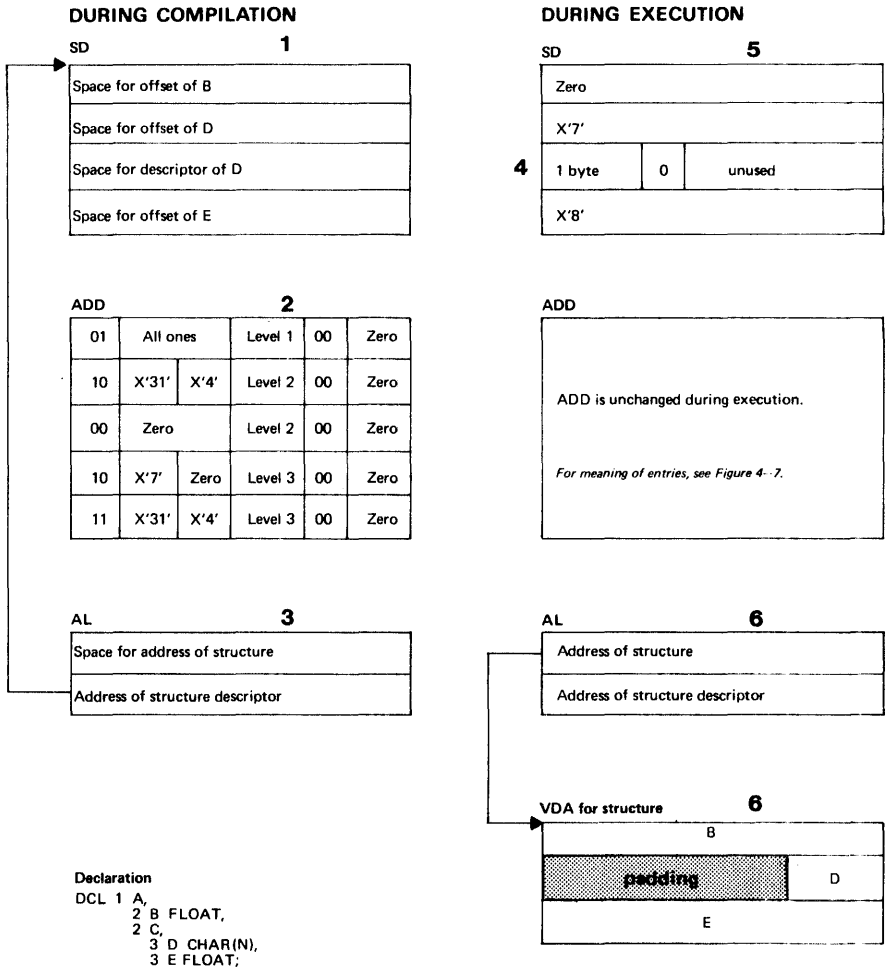


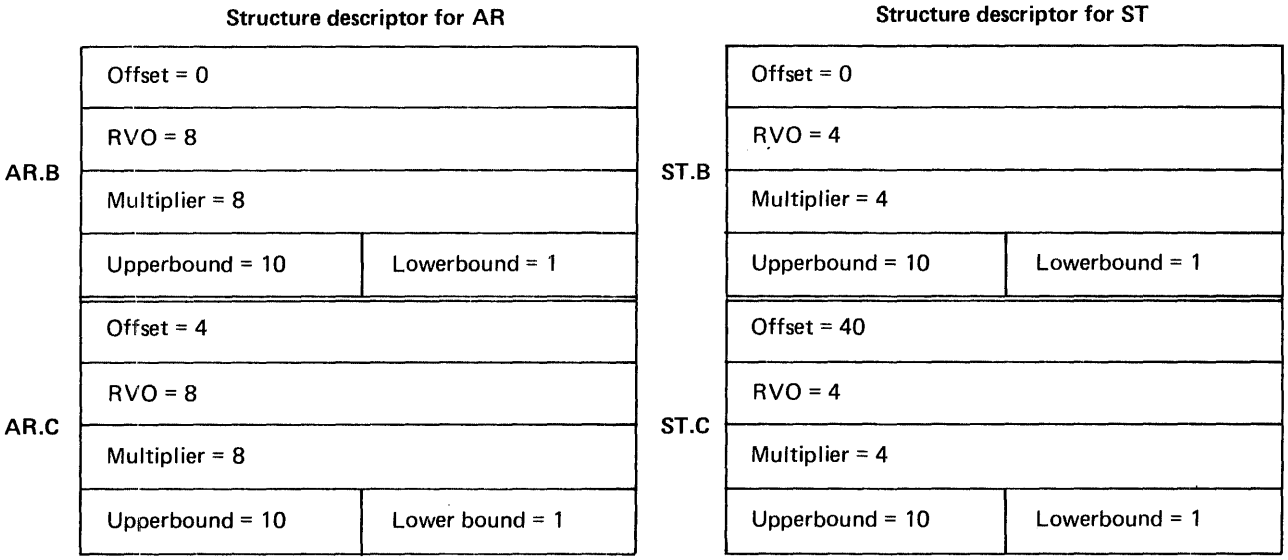
Figure 4.8. Example of handling a structure containing an adjustable extent

Array of structures

```
DCL 1 AR(10),
      2 B,
      2 C;
```

Structure of Arrays

```
DCL 1 ST,
      2 B(10),
      2 C(10);
```



Note: Descriptors are identical apart from multiplier RVO offset values

Figure 4.9. Structure descriptor for arrays of structures and structures of arrays

FEDs (input/output DEDs): fall into five classes

1. A,B, and control format FEDs have four bytes.

2. E and F format FEDs are six bytes long.

3. Pictured arithmetic FEDs consist of four bytes followed by the pictured arithmetic DED.

4. Pictured character string FEDs consist of four bytes followed by the pictured character string DED.

5. C format FEDs are four bytes plus the two constituent FEDs that make up the complex item. They are used for complex data.

The first two bytes of any DED are the look-up byte and the flag byte. Taken together, they define the data type and permit a receiving routine to determine if it needs to look further into the DED for more information. The general format of DEDs is shown in figure 4.10. Full details are given in appendix A.

SYMBOL TABLES AND SYMBOL TABLE VECTORS

Data-directed I/O statements, and the CHECK condition, require the names of variables to be available throughout execution. Normally, such names are not used after compilation. When required during execution, these names are held in control blocks known as symbol tables. Symbol tables hold the name of the variable, its address, and the address of its DED plus certain other information (see appendix A).

GET DATA and PUT DATA statements without a data list, and SIGNAL CHECK statements when there is no check list, imply that the names of all variables known at that point in the program must be available. The necessary information is held in a further control block known as the symbol table vector. The symbol table vector holds the addresses of symbol tables arranged in order of program blocks, commencing with the main procedure block. The symbol table vector consists of a series of fullword fields. These fields contain either the address of a symbol table, a fullword of zeros, or a further address within the symbol table vector. The end of entries for variables declared in each block, is followed by a fullword of zeros, which in turn is followed by the address in the symbol table vector where entries for the encompassing block begin. If there is no encompassing block, another word of zeros marks the end of the vector.

Figure 4.11 shows the relationship between variables, symbol tables, and the symbol table vector.

Data-directed I/O modules, and the CHECK module, use symbol tables and symbol table vectors in the following ways.

GET DATA (A,B,C), PUT DATA (A,B,C), SIGNAL CHECK (A,B,C): In all these cases, the addresses of the symbol tables for A, B, and C are passed to the appropriate library module.

GET DATA, PUT DATA, SIGNAL CHECK: When no data or check list is included in the statement, the library is passed the address of the start of the associated block entries for the symbol table vector. By following the symbol table vector, it is possible to access the names of all the variables known in the block.

The contents of symbol tables vary according to the storage class of the variable. The method used for holding the address, and other information, is given in appendix A. For internal variables, symbol tables are held in static internal storage. For external variables, symbol tables are held as separate control sections in static external storage. The name of each control section is the name of the associated variable followed by an X. Thus the control section for the external variable B would be BX. Such a control section would also contain the DED of the variable (or DEDs if the variable was a structure).

String DED

Look-up byte	Flag byte	Not used

Arithmetic DED

Look-up byte	Flag byte	Precision	Scale

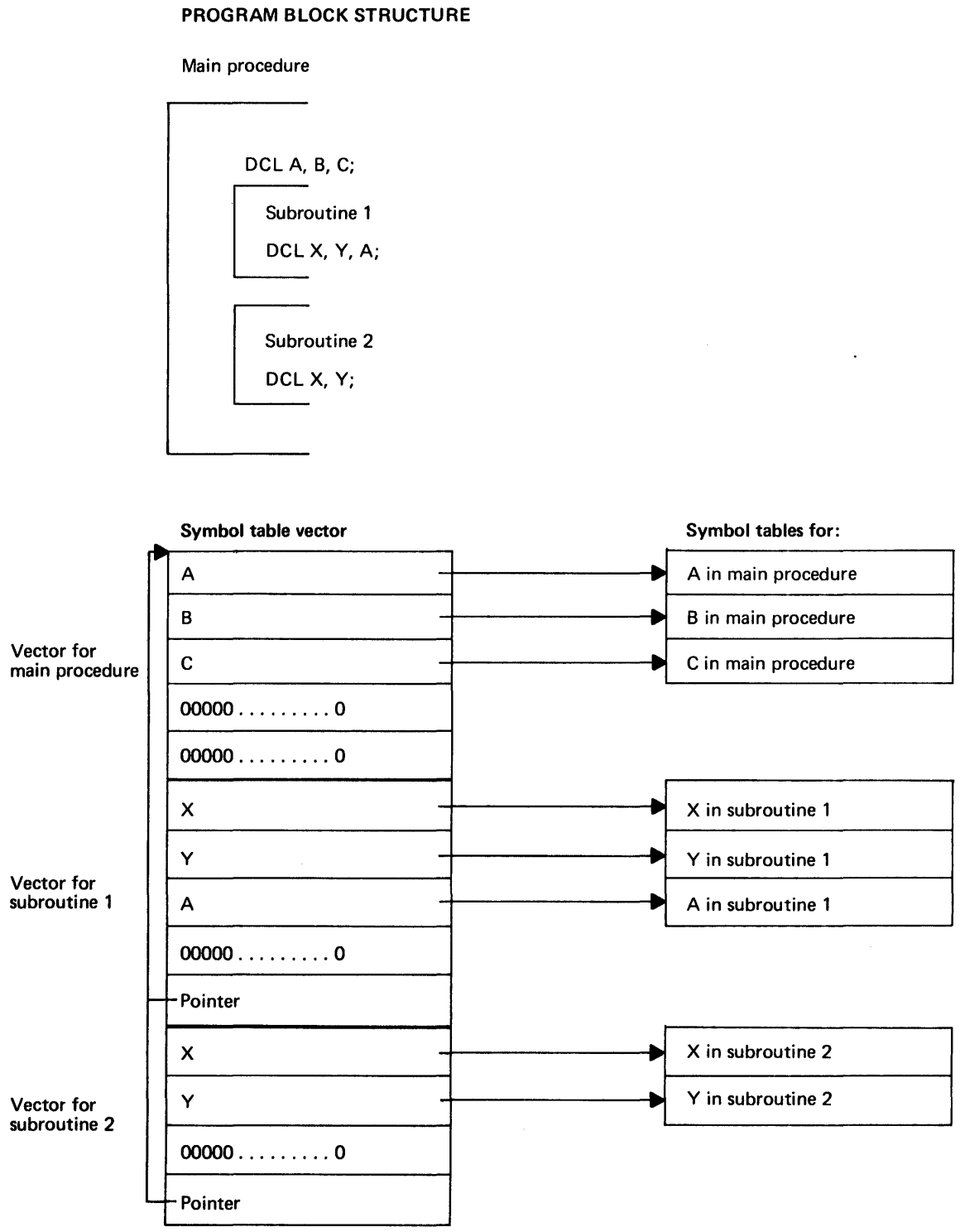
Pictured string DED

Look-up byte	Flag byte	Length of string
Length of string without/insertion characters		Translation of picture
specification into internal format (one byte per character)		

Pictured arithmetic DED

Look-up byte	Flag byte	Precision	Scale
Length of picture	Length of data	Mantissa byte	Exponent byte
Translation of picture specification into internal format (at least one byte per character)			

Figure 4.10. Format of DEDs



The symbol table vector is built up on a block by block basis, the last entry for each block being a word of zeros followed by a pointer to the first entry for the encompassing block. This mechanism allows for multiple declarations of names.

Figure 4.11. Symbol tables and symbol table vectors

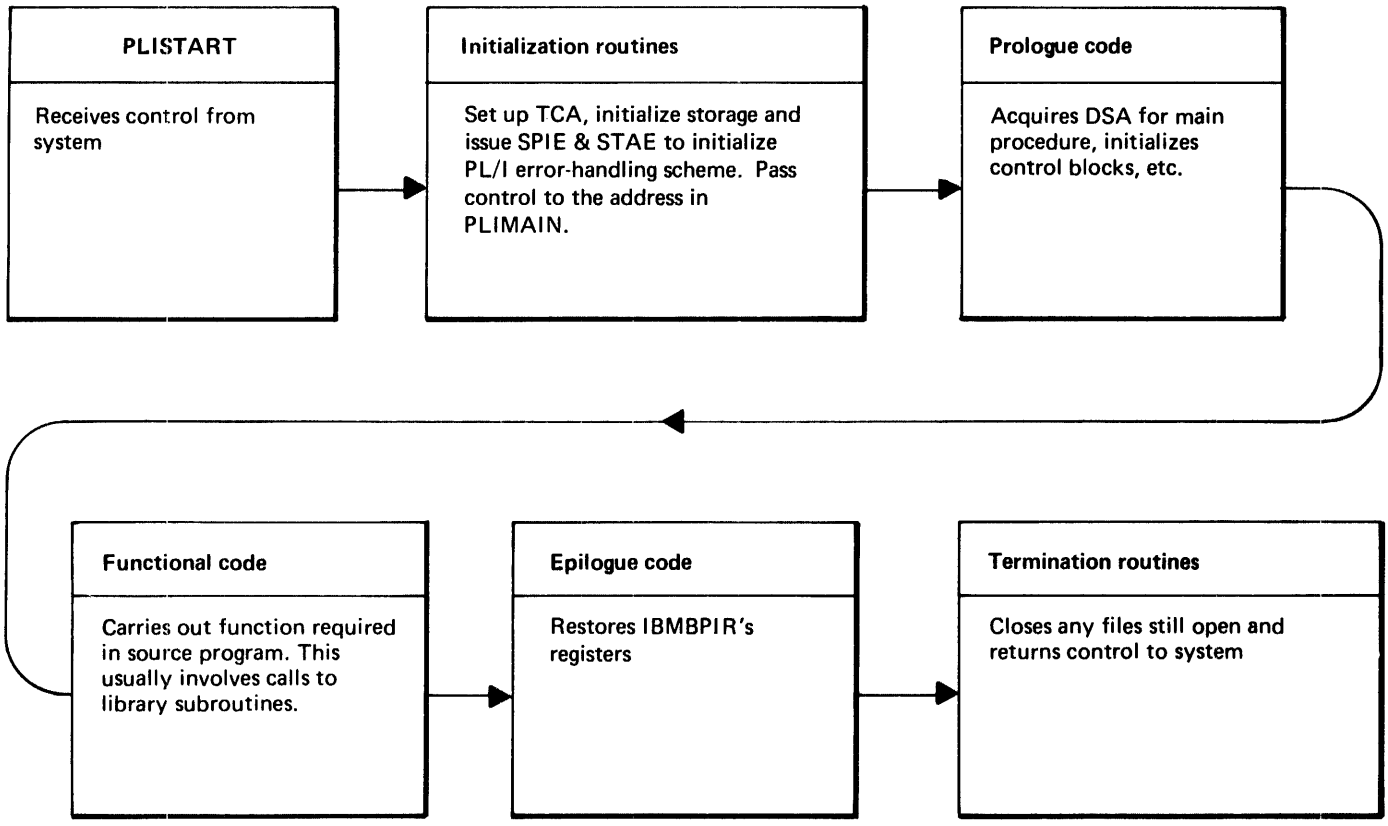


Figure 5.1. Flow of control during execution

Chapter 5: Object Program Initialization

Before the output from the compiler can be executed, it must be link-edited, and the PL/I environment must be set up. This chapter briefly describes the effects of link-editing, the manner in which the program is entered, and the initialization process that sets up the PL/I environment. Initialization for multitasking programs is explained in chapter 14. It also gives a brief description of the program management area; a control area set up during program initialization.

Link-editing

The functions and use of the linkage editor program are described in the publication OS Linkage Editor and Loader. This chapter describes the effects of link-editing on the PL/I program. The linkage editor combines the various control sections generated by the compiler and resolves addresses within these control sections. The linkage editor also incorporates into the executable program phase all library modules that are called from compiled code, and a number of other library modules that are required either because they in turn are called by the library modules called by compiled code, or because they are needed for program management. A major module used in program management is the error-handling module, IBMBERR. An external reference to this module is contained in the PL/I initialization routine, IBMBPIR. An external reference to IBMBPIR is included in the control section PLISTART which is generated by every compilation and nominated as its entry point. PLISTART contains an external reference to the control section PLIMAIN (which holds the address of the start of the main procedure).

One of the features of the linkage editor is that it does not accept more than one control section with the same name; the second use of the name is ignored. As a result of this, only one PLISTART and one PLIMAIN is generated for each executable program phase. This allows two or more PL/I main procedures to be link-edited together. The procedure that receives control will be the first that is passed to the linkage editor, because it will be the PLISTART and PLIMAIN of this procedure that are included in the executable program. This feature is also used to handle data declared EXTERNAL. Control sections for each such data item are generated by all programs in which the data is declared. Only one of these is resolved.

Note: The entry statement cannot be used to pass control to a specified PL/I program as entry must be made through PLISTART.

The PLIMAIN control section is not generated by the compiler if the PL/I source program does not contain the MAIN option. However, a control section named PLIMAIN is included in the initialization module IBMBPIR. This control section contains the address of code that calls the module IBMBPEP, which puts out a message saying there is no main procedure, after which the program is terminated.

Program Initialization

Code is compiled by the PL/I Optimizing Compiler on the assumption that various control blocks will have been set up and that certain registers will point to them when the program is entered. This arrangement of control blocks and registers is known as the PL/I environment.

The most important factors affecting the PL/I environment are the following:

1. An area for the allocation of PL/I dynamic storage should be available. This area is known as the initial storage area (ISA).

2. A dynamic storage area (DSA) should exist. This will give the address of the start of the area available for dynamic storage allocation and will act as a save area for the calling routine's registers.

3. A task communications area (TCA) should exist. The TCA acts as a central communications area for the program, holding addresses of various storage- and error-handling routines, and control blocks. The TCA also contains a number of flags and other fields.

4. Program checks should be passed to the PL/I error-handling module IBMBERR.

5. Pre-formatted DSAs should exist for certain library routines. These pre-formatted DSAs are known as library

workspace (LWS).

6. A space should be available for any condition built-in function values (ONCHAR, ONSOURCE, etc.) should a PL/I interrupt occur. This space is known as an ON communications area (ONCA). As the condition built-in functions have default values, an area to hold the default values is required. This is known as the dummy ONCA.

7. Register 12 should point at the TCA, and register 13 should point to the DSA.

The resident program initialization routine IBMBPIR, and the transient routine IBMBPII, which it calls, acquire the ISA, and set up the various control blocks in an area of the head of the ISA known as the program management area. The contents of the program management area are described later in this chapter.

The default ISA size and other options are controlled either by the system default module IBMBOPT or by specifying an external variable called PLIXOPT within the program.

The use of initialization routines obviates the need for special code in main procedures, and allows two procedures with the MAIN option to be used in the same program.

As shown in figure 5.1, the initialization routine IBMBPIR is reentered after the execution of compiled code. This is done by the standard action of the epilogue code. The registers of IBMBPIR are stored in the dummy DSA by the prologue code, and restored by the epilogue code. When terminating the program, IBMBPIR calls a transient library routine, IBMBPIT, to handle the majority of the termination functions.

INITIALIZATION AND TERMINATION ROUTINES

Three routines are used in initialization and termination. They are:

1. IBMBPIR - resident library initialization/termination routine.

2. IBMBPII - transient library initialization routine.

3. IBMBPIT - transient library termination routine.

The use of transient routines reduces the space overhead required.

The resident routine, IBMBPIR, is a short control routine. The major functions are carried out by the transient routines. However, IBMBPIR contains a number of housekeeping subroutines, including code to handle GOTO out of block in certain abnormal situations, and the STAE exit subroutine. These are described in chapters 6 and 7 respectively.

Resident Initialization/Termination Routine IBMBPIR

IBMBPIR has three entry points. One of these is for use by the supervisor; the other two are for use by problem programs written in languages other than PL/I. The main difference between the entry points is the parameters that are expected. The entry points are:

1. IBMBPIRA - used when entry is made from the system.

2. IBMBPIRB - for use by non-PL/I callers who wish to accept PL/I default ISA size.

3. IBMBPIRC - for use by non-PL/I callers who wish to nominate the length and, optionally, the address of dynamic storage used by PL/I.

Entry points B and C will be used by programmers specifying PLICALLA and PLICALLB respectively. (PLICALLA and PLICALLB are entry points in the control section PLISTART.) Using PLICALLA results in control being passed to IBMBPIRB. Using PLICALLB results in control being passed to IBMBPIRC.

IBMBPIRA and IBMBPIRC can be passed a number of parameters related to program management. These include ISASIZE and REPORT. IBMBPIR assumes that all parameters preceding a slash (/) are program management parameters. All main procedure parameters must, therefore, be preceded by a slash, otherwise they are taken to be parameters for IBMBPIR.

Entry point IBMBPIRC can be passed a parameter list that contains (in the second and third words) the length and, optionally, the address at which the ISA is to begin. The ISA size and address are passed to IBMBPII.

Entry point IBMBPIRB cannot accept any parameters; the default ISA size is always given. (See below, under the heading "Acquiring the ISA.")

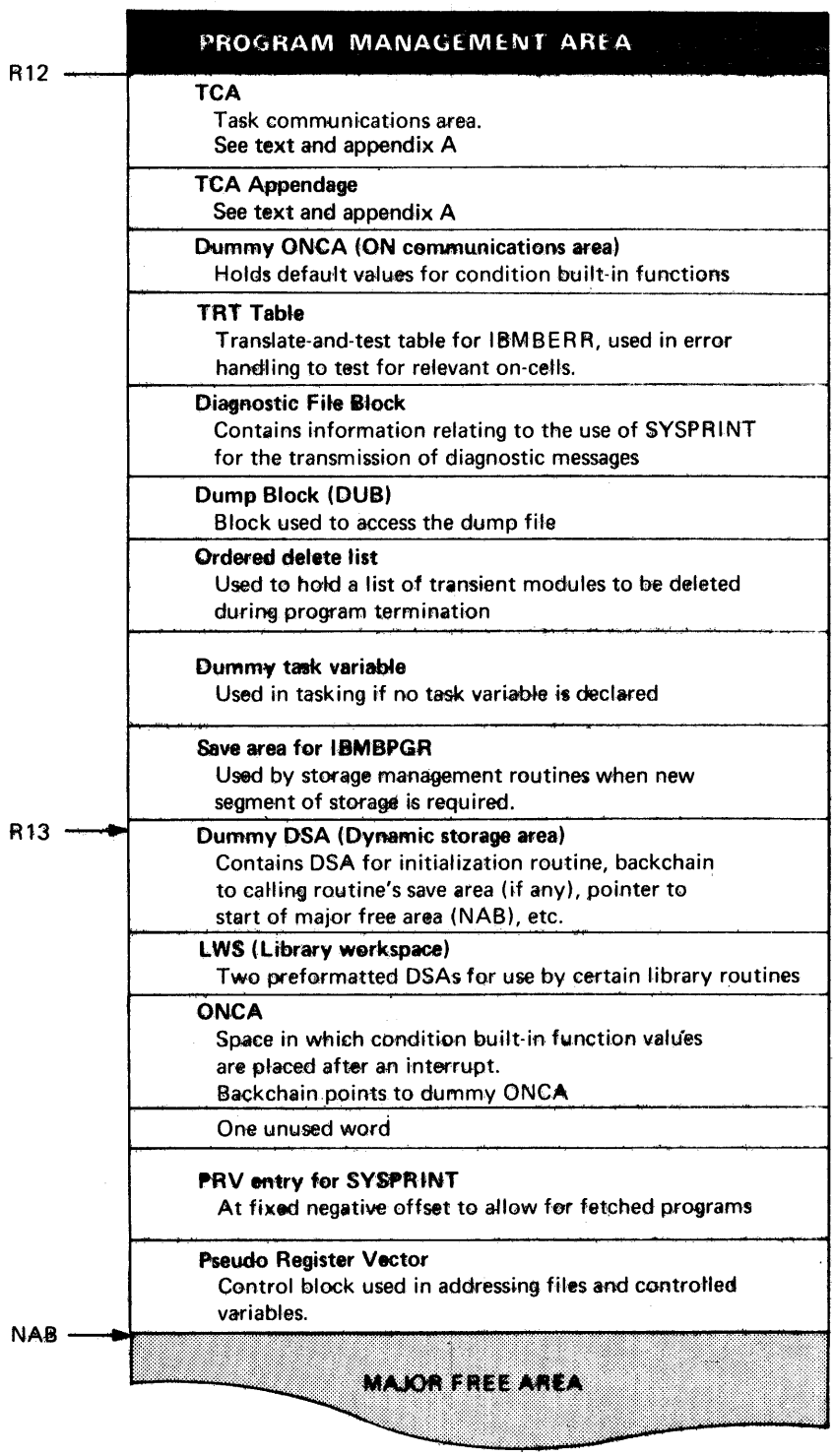


Figure 5.2. Program management area

The Process of Initialization

When IBMBPIR is called to initialize the program, it acquires workspace, then loads and calls IBMBPII. IBMBPII carries out the actions described below.

Handling Execution Time Options

IBMBPII first analyzes the execution time options. Execution time options, which were also known as program management parameters, can be specified in the following three ways:

1. As parameters of the EXEC statement,
2. From an external variable called PLIXOPT in the PL/I program.
3. From the default module IBMBXOPT which is set up during compiler generation.

All three sources may exist and the options are merged from them. IBMBPII first loads the default module IBMBXOPT. It then searches for a control section called PLIXOPT which will have been produced by the compiler if an external variable called PLIXOPT was declared in the program. Any options specified in PLIXOPT are then merged with those in IBMBXOPT with the values in PLIXOPT overriding those in IBMBXOPT. The process is then repeated with any execution time options specified in as parameters in the EXEC statement. When the execution time options have been sorted out, IBMBPII carries out the actions described below.

Acquiring the ISA

The method of acquiring storage for the ISA depends on the entry point of IBMBPIR used.

If entry point C is used, and both the ISA size and address have been passed, no further action need be taken.

If the ISA size has been passed, to either entry point C or entry point A, a GETMAIN macro is issued for the amount of storage requested.

If no ISA size has been specified, the default action is taken. The default action is to obtain all the available storage. The high-address half of this storage is then freed, and the lower half retained as the ISA. If the resulting figure is not large enough to hold the program management area, an area large enough for the program management area is obtained.

If there is not enough space for the ISA size requested, or if the defaults do not provide enough space for the program management area, the action described below under the heading "Error Situations" is taken.

Initialization of the Program Management Area

The program management area is set up at the low address end of the ISA. IBMBPII initializes the various control blocks. These are shown in figure 5.2. Their functions are described below under the heading "The Program Management Area."

The storage management routine is loaded, and the addresses of its various entry points are placed in the TCA. If a storage report is requested, module IBMBPGD is loaded; otherwise, module IBMBPGR is loaded.

Initializing PL/I Error Handling

The PL/I error handling scheme handles all program checks, and attempts to handle ABENDs. The address of the old PICA is saved in the TCA so that the previous SPIE may be restored during program termination, and SPIE and STAE macro instructions are issued to set up the PL/I error handling scheme.

The SPIE macro specifies entry into entry point A of the error handling module IBMBERR. The STAE macro specifies entry into the STAE exit subroutine in IBMBPIR. (This subroutine loads the ABEND analyzing module IBMBPES.) A full description of the PL/I error handling facilities is given in chapter 7.

When the program management area has been initialized, and the SPIE and STAE macro instructions have been issued, IBMBPII returns control to IBMBPIR.

IBMBPIR checks that the return has been normal and, if so, points register 1 at the parameters for the main procedure, and calls the procedure whose address is held in the control section PLIMAIN.

NOSPIE and NOSTAE options: If NOSPIE is

specified in the parameters passed to IBMBPIR no SPIE macro is issued by the initialization routine. This allows an installation to specify its own method of handling program check interrupts. Similarly if NOSTAE is specified a STAE macro will not be issued.

Error Situations

If there is insufficient storage available to meet the requested ISA size, IBMBPII calls IBMBPEP, which puts out an "INSUFFICIENT MAIN STORAGE" message. IBMBPII then returns to IBMBPIR, requesting it to free the storage acquired, and terminate the program.

If no PL/I main procedure has been provided, and there is no alternative PLIMAIN control section provided by the user, a control section PLIMAIN in IBMBPIR will have been link-edited. When control is passed to the address contained in this control section, an error module is called. A 'NO MAIN PROCEDURE' message is generated, and the program is terminated.

The Process of Termination

When the main procedure is complete, epilogue code for the main procedure returns control to IBMBPIR, passing to it the address of the main procedure DSA. If the termination is normal, IBMBPIR restores the value of register 13 to that passed to it in register 0. IBMBPIR then sets flags in the TCA indicating that the program is terminating, and calls the error handler to raise the FINISH condition. If there is no GOTO from a FINISH on-unit, the error handler will return to IBMBPIR using the GOTO-out-of-block mechanism. The flags set in the TCA to indicate program termination are tested and, as they are set, control is returned from the GOTO code in the TCA to the abnormal-GOTO subroutine in IBMBPIR. This routine handles any outstanding housekeeping problems. Exit DSAs are correctly terminated and, because the program termination flags are set, all files are closed by calling IBMBOCL. Control is then returned to the termination routine, IBMBPIR. (A full discussion of the GOTO-out-of-block mechanism and its implications is given in chapter 2.)

IBMBPIR then calls IBMBPIT to complete the housekeeping. STAE and SPIE macro instructions are issued to restore the error handling situation, and control is returned to the caller.

THE PROGRAM MANAGEMENT AREA

A diagram of the program management area is shown in figure 5.2. It shows the situation when the compiled program is called. The various fields in the program management area are shown in detail in appendix A. A brief description of their use is given below.

Task Communications Area (TCA)

The TCA is the central communications block used throughout the program. It is used to address the error-handling and storage-management routines, and to point to the current segment of dynamic storage.

A field-by-field description follows.

Flags — Indicate that an abnormal GOTO out of block may take place (see below). Also indicate that certain special error conditions may arise.

BOS — The pointer that points to the beginning of the current segment (see chapter 6).

EOS — The pointer that points to the end of the current segment (see chapter 6).

Address of external save area:

> The address of the save area for the calling routine, if IBMBPIR was not called from the control program.

Address of translate-and-test table for IBMBERR:

> See below, under heading "Translate-and-Test Table."

Address of TCA appendage

Address of save area for IBMBPGRC and IBMBPGRD (see below)

Open file chain:

> Used when closing files at end of job

Address of IBMBPGRD:

> Stack overflow routine for VDAs (see chapter 6)

Address of the diagnostic file block (see below)

PL/I and user return code:

A standard area to keep these codes.

Address of flow statement table:

This is used to address the flow statement table which holds statement numbers for use during execution.

Address of tab table:

The address of a table of tabulator positions used in list-directed output.

Address of FLOW module:

The address of the module used to implement the compiler FLOW option.

Shared library transfer vector addresses:

Used when accessing PL/I library modules in the link-pack-area.

Address of PRV initialization word:

Used to access word set in PRV when files are closed.

Address of control task service routines:

Used to access service routines in multitasking.

Address of storage-handling routines:

Entry points to IBMBPGR that get non-LIFO storage, free non-LIFO storage, and acquire a new segment for LIFO storage (see chapter 6).

Address of IBMBERRB

Address branched to after a software-detected interrupt occurs (see chapter 7).

Environment descriptor:

Identifies release of libraries being used.

Code for GOTO out of block:

Whenever a GOTO out of block occurs, or could potentially occur because of the value of a label variable, compiled code branches to this code in the TCA.

The function of this code is described in chapter 2, under the heading "Handling Flow of Control."

Address of get-control routine:

Routine used in multitasking (see chapter 14)

Address of free-control routine:

Routine used in multitasking (see chapter 14)

Address of ENQ SYSPRINT routine:

Library routine used in stream I/O (see chapter 9)

Address of DEQ SYSPRINT routine:

Library routine used in stream I/O (see chapter 9)

Address of WAIT module:

Address of IBMBJWT, the module used to execute the WAIT statement

Address of COMPLETION pseudovariable module

Address of event assign module

Address of priority routine

Address of ENQ and DEQ routines:

Used for enqueuing and dequeuing files other than SYSPRINT.

TCA Implementation Appendage

The TCA implementation appendage (TIA) is addressed from the standard part of the TCA. Its contents are as follows:

Address of the byte beyond the ISA(TISA):

This holds the address beyond the end of the partition and is necessary because EOS gets altered when non-LIFO dynamic storage is allocated.

Address of old PICA (TAPC):

Used to restore SPIE to that which existed when the PL/I program was called.

Address of interrupt handler (TERA):

This is the address to which the branch is made after a program check interrupt (see above) has occurred.

Interrupt mask and flags (TINM)

Wait information table (WIT) chain header (TWTW):

Start of the chain indicating which events are being waited-on in the task.

Anchor for chain of exclusive blocks (TEXF)

Used when handling exclusive files

Address of last free area (TLFE):

Address of last free area of non-LIFO storage on the free area chain: used as a starting point when searching the chain.

Address dump block (TDUB):

Used when a PLIDUMP is being executed.

Address of dummy DSA:

Used, when abnormally terminating the program, to restore IBMBPIR's registers. This allows IBMBPIR to be reached should the DSA chain be overwritten.

Address of get-library-workspace routine:

This is part of the resident library module IBMBPIR and is used to get a new allocation of library workspace and an ONCA. This routine is called after interrupts and during program initialization (see chapter 3).

Address of extended float simulator (TASM):

Used on machines that do not have the extended floating-point instructions to handle extended floating-point data.

Name of extended float simulator (TSNM):

Used to hold the name of the extended float simulator, so that it can be invoked if required.

Save Area for IBMBPGR

This area is used as a DSA for IBMBPGR, the routine entered when there is not enough room for a further DSA in the current segment of the LIFO stack. Both DSAs in library workspace may be in use when IBMBPGR is required, and there may be nc caller's save area because a DSA has not yet been acquired. Consequently, IBMBPGR has a save area reserved in the program management area.

Dummy ONCA

The dummy ONCA holds default values for the condition built-in functions. These will be supplied if they are requested either when no interrupt has occurred, or when nc interrupt with the requested condition built-in function value has occurred. There is a chain back through all ONCAs to the dummy ONCA. (See chapter 7.)

Translate-and-Test Table

The translate-and-test table contains code used in error handling to identify relevant on-cells. (See chapter 7.)

Dump File Block

This is space used during the execution of PLIDUMP to hold the DCB and other information for the dump file.

Loaded Module or Ordered Delete List

This is a list of modules that are deleted by IBMBPIR during program termination. Certain transient modules that are not deleted by other methods place their name in this list to ensure that they are deleted when the program is terminated.

Dummy Tasks and Event Variables

These are included in the program management areas to allow the use of the STATUS and PRIORITY built-in functions in non-multitasking programs, and to allow multitasking programs to operate if no task

or event variables are explicitly declared.

Diagnostic File Block

The diagnostic file block holds information used by the error-message modules. This includes the address of the SYSPRINT transmitter.

Dummy DSA

The dummy DSA acts as a save area for the registers of the initialization routine IBMBPIR, and an end to the chain of DSAs when a search through blocks is being made, as, for example, when searching for a relevant established on-unit (see chapter 7). The dummy DSA has a bit in its flag byte to indicate that it is a dummy. The dummy DSA contains a NAB (next available byte) pointer enabling the main procedure to obtain a DSA in the LIFO stack.

Library Workspace (LWS)

This consists of two pre-formatted DSAs that are used by certain of the library modules. (See chapter 3.)

ON Communications Area (ONCA)

The ONCA is an area where compiled code or library routines can store or read any condition built-function values that may be required. (See chapter 7.)

Pseudo-Register Vector

This is used in addressing files and controlled variables. (See chapter 2.)

Multitasking

The program initialization process for a multitasking environment is described in chapter 14.

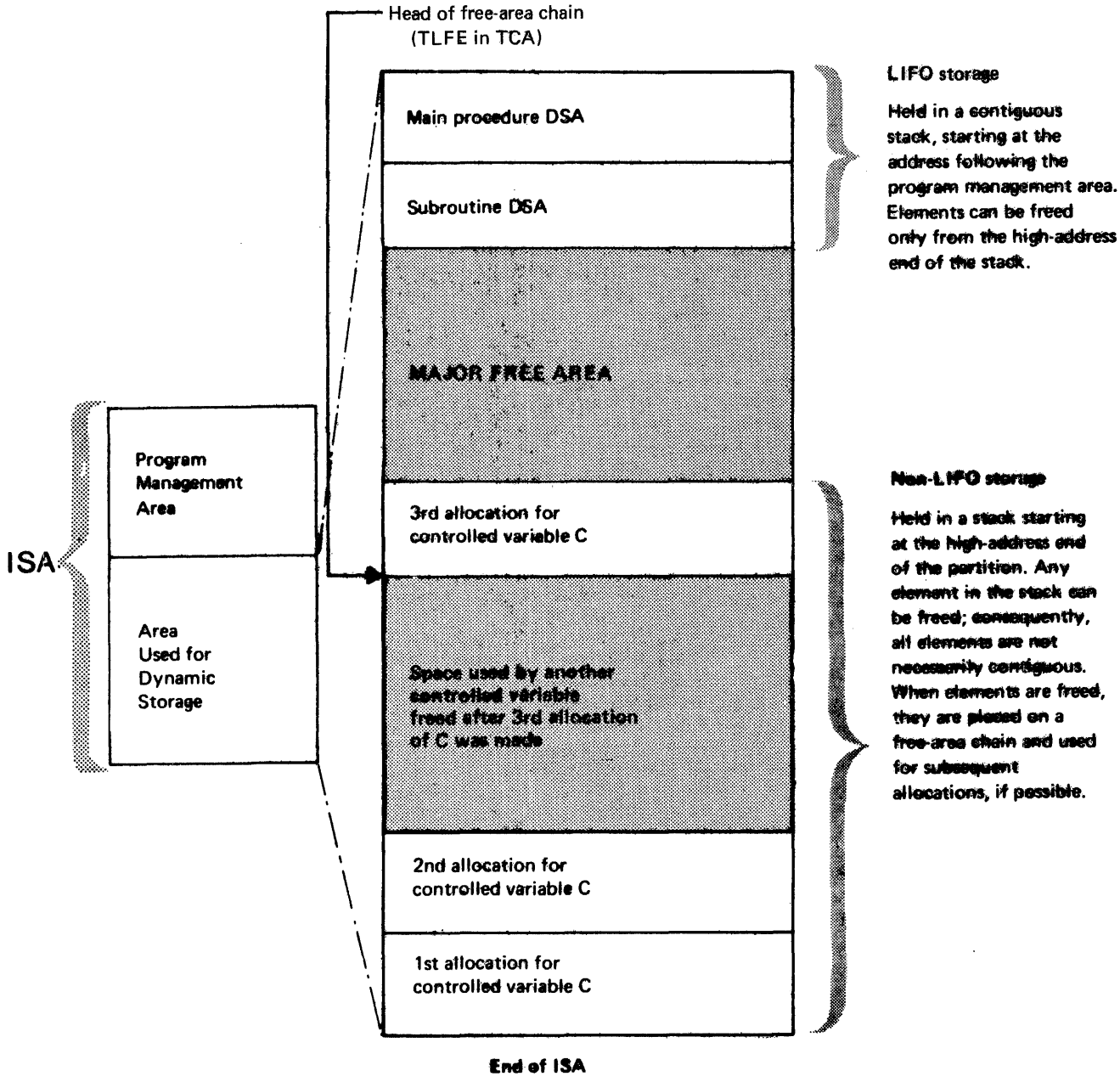


Figure 6.1. Use of storage in the ISA

Chapter 6: Storage Management

The OS PL/I optimizing compiler allows the user to specify the working storage area he wishes to use, by a parameter known as ISASIZE. When this parameter is specified, the initial storage area (ISA) is set up to the size indicated. This is done by issuing a GETMAIN macro instruction for the required amount of storage.

If the ISASIZE parameter is not specified, or if the ISA size specified is greater than the size of the region, default action is taken. The default action is to issue a variable GETMAIN instruction for the largest amount of storage possible. Half of this storage is allocated to the ISA and the remainder is freed for possible future use by the program or by the system.

The allocation of the ISA is handled in the program initialization module IBMBPII. The procedure for tasking is slightly different and is described in the section on multitasking at the end of this chapter. The initial storage area (ISA) is used for various functions during execution. The start of the ISA is used as the program management area. The program management area contains a number of housekeeping fields and is set up by the initialization routines. (See chapter 5.) The remainder of the ISA is used for PL/I dynamic storage allocation.

TYPES OF DYNAMIC STORAGE REQUIRED

The requirement for dynamic allocation and freeing of storage is inherent in the language. Automatic variables are allocated and freed on a block-by-block basis. Controlled and based variables can be allocated and freed by appropriate PL/I statements. Storage is also obtained dynamically for workspace, and compiler-generated temporary values.

Dynamic storage can be conveniently divided into two classes.

1. That which is allocated and freed on a last-in/first-out (LIFO) basis.
2. That which is not.

The first class is known as LIFO dynamic storage and the second class as non-LIFO dynamic storage.

Certain other storage is also acquired dynamically. This is storage for transiently loaded library modules and input/output buffers. This storage is acquired and freed directly under system control. Routines wishing to load a transient module issue a LINK, LOAD, or XCTL macro instruction. When the transient module is to be freed the controlling library module issues the necessary macro instruction.

Contents of LIFO (Last-In/First-Out) Storage

Two kinds of storage area are allocated in LIFO storage. They are dynamic storage areas(DSAs) and variable data areas(VDAs). A DSA is allocated for every procedure or block and contains:

- The System/360 standard save area.
- Certain standard housekeeping fields.
- All automatic variables and compiler-generated temporaries whose length is known during compilation.

A diagram of the standard section of a DSA is shown in appendix A.

VDAs are acquired for all other allocations of LIFO dynamic storage. These include:

- Storage for automatic variables and compiler-generated temporaries whose length is not known until execution. (X CHAR(N), for example.)
- Workspace for certain library modules.
- Allocations of library workspace (LWS) after the occurrence of an interrupt.

Contents of Non-LIFO Storage

Non-LIFO storage is used for the following:

- Controlled variables.
- Those based variables that are allocated by the ALLOCATE statement, (provided that they are not allocated in an automatic or static AREA).

Dynamic Storage Allocation

The principle used in dynamic storage allocation is to allocate LIFO storage from the low-address end of ISA, starting at the first 8-byte boundary beyond the program management area, and to allocate non-LIFO storage from the high-address end of the ISA. Between the areas of LIFO and non-LIFO storage is an unused section known in this publication as the major free area. (See figure 6.1.)

The last element in the LIFO stack is always the first to be freed and consequently can always be amalgamated with the major free area. This is not always the case with non-LIFO storage. When an item not contiguous with the major free area in the non-LIFO stack is freed, it is placed on a free-area chain whose head is anchored in the TCA. Attempts are always made to use areas on this chain when further allocations of non-LIFO storage are made.

Allocations of LIFO storage are made by testing to see if there is enough space in the major free area. If there is not enough space, an attempt is made to use an area on the free-area chain. When an area on the free-area chain is used, it is known as a new segment of the LIFO stack.

If there is no space either in the major free area or on the free area chain, then a GETMAIN macro instruction is issued to obtain new storage. For LIFO storage this will be set up as a new segment of the LIFO stack and the necessary housekeeping fields will be placed at its head.

Fields Used in Storage Handling

To keep track of the storage allocated and freed, a number of pointers are used. These are:

- The beginning-of-segment pointer (BOS).
- The end-of-segment pointer (EOS).
- The next-available-byte pointer (NAB).
- The free-area chain pointer TFLE.
- A pointer to the byte beyond the end of the ISA (TISA).

The beginning-of-segment pointer (BOS) is initially set during program initialization to point to the start of the ISA. It is not altered unless a new segment of storage is acquired. BOS always points to the start of the current storage segment. BOS is held at offset 8 from the head of the TCA, and is addressed from register 12.

The end-of-segment pointer (EOS) is initially set during program initialization to point to the end of the ISA. However, it is updated, when non-LIFO storage is allocated, to point to the end of the major free area. EOS is held at offset X'C'(12) in the TCA, and is addressed from register 12.

The next-available-byte pointer (NAB) is held in every DSA and points to the first 8-byte boundary contiguous with unused storage. This address is the start of the major free area. The current NAB is held in the most recent DSA and addressed from offset X'4C'(76) from register 13. As register 13 is altered every time a DSA is acquired, the value in a NAB pointer need only be altered when a VDA is freed or acquired. Previous NABs are automatically restored when register 13 is pointed to a previous DSA.

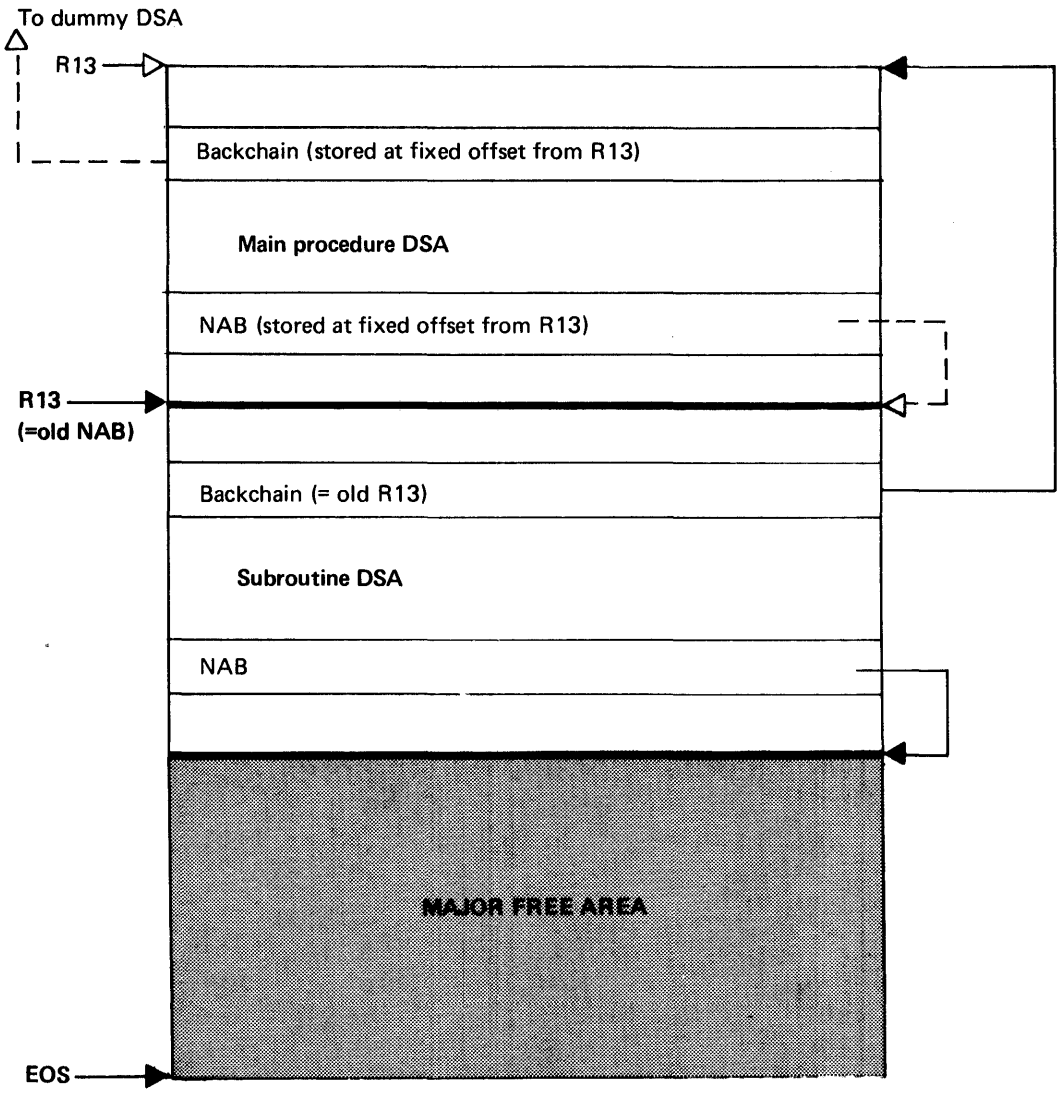
The pointer to the byte beyond the ISA (TISA) is used to keep track of the end of the ISA.

The first byte of BOS, EOS, and NAB contain segment numbers ("FF" for the ISA). The use of these numbers is explained under "Acquiring a New Segment."

The free area chain pointer TLFE. The free-area chain includes those elements of non-LIFO dynamic storage that have been freed but that could not be amalgamated with the major free area. The start of the chain is held at offset 8 in the TCA appendage in a field called TLFE. TLFE points to the element with the highest address.

ALLOCATING AND FREEING LIFO STORAGE

Allocating and freeing LIFO storage is handled by compiled code or by the particular library module that requires the space. The allocation is done in the manner used by the prologue code shown in chapter 2. Freeing is done in the manner used by the epilogue code, which is also shown in chapter 2. Before allocating the storage, a test is made to see if there is enough space in the major free area for the new allocation. For reasons explained later under the heading "Acquiring a New Segment," this test is carried out by logical arithmetic. If there is not enough space, entry to one of the segment-handling entry points of the transient library module IBMBPGR is made. The entry point



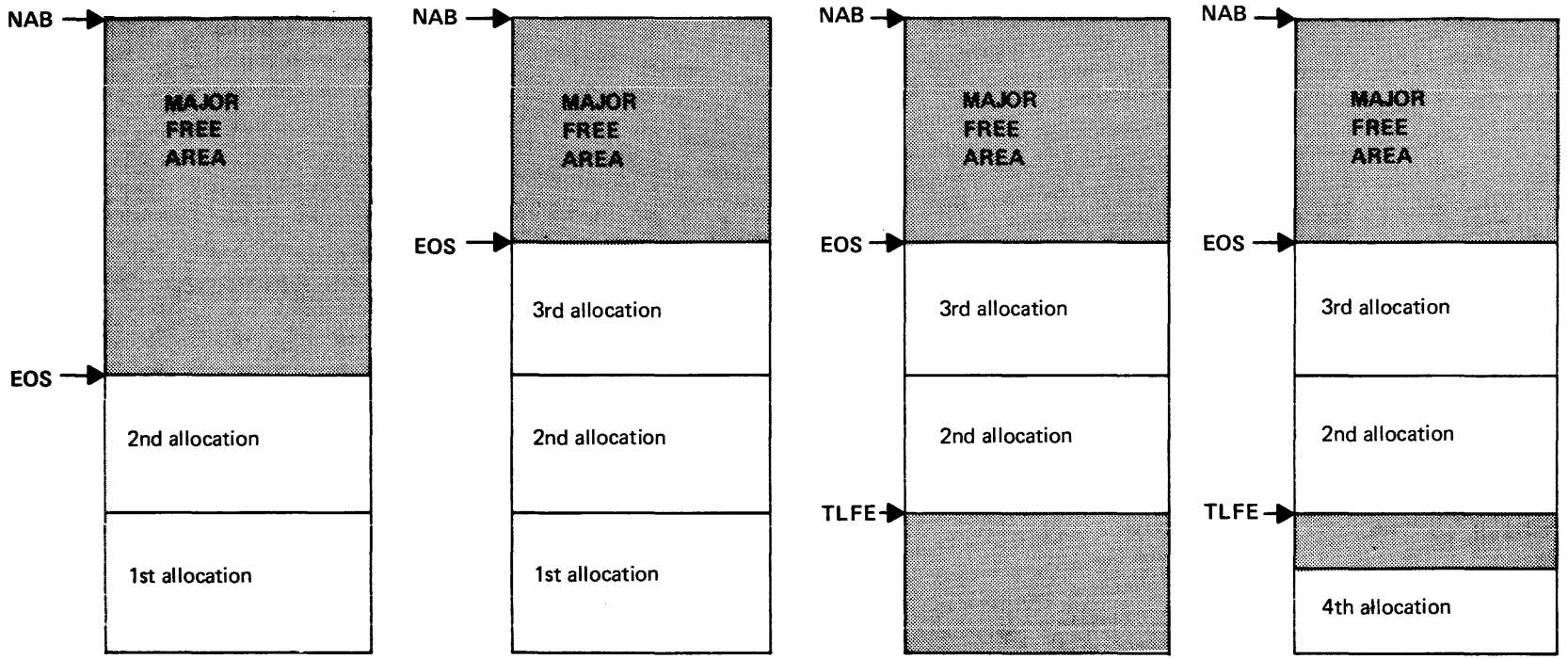
Allocating a new DSA

1. Test if major free area large enough for new DSA. If not call IBMBPGRC.
2. Store R13 at fixed offset from old NAB to act as backchain.
3. Load R13 with address of old NAB.
4. Store new NAB at fixed offset from register R13.

Freeing a DSA

1. Load register 13 with current backchain address. Since the NAB and backchain fields are always addressed from register 13, the previous values are automatically restored.

Figure 6.2. Principles involved in allocating and freeing LIFO storage



Initial situation

Two contiguous allocations of non-LIFO storage

New allocation

1. Free-area chain? No. (TLFE in TCA=0)
2. Allocate by altering EOS pointer.

Free 1st area

1. Is area next to major free area? No.
2. Is area next to an area already on free-area chain? No.
3. Place area on free-area chain.

Further allocation

1. Free-area chain? Yes.
2. Find smallest area that will hold new allocation. Allocate at high-address end, leaving remaining area on free-area chain.
3. Alter length field at head of remaining area.

Figure 6.3. Principles involved in allocating and freeing non-LIFO storage

used depends on whether a VDA or a DSA is being acquired. The allocation of LIFO storage involves accessing the current value of NAB. This gives the address of the start of storage to be used. A new NAB value is calculated, addressing the byte beyond the end of the new allocation. Register 13 is pointed at the old NAB value and the new NAB value placed at offset X'4C' from register 13. Freeing the storage is done by restoring register 13 to the previous value. Figure 6.2 illustrates the principles involved. Before allocating the storage, a test is made to see if there is enough space for the allocation. When there is insufficient space for a LIFO storage allocation, a new segment is acquired. (See below.)

ALLOCATING AND FREEING NON-LIFO STORAGE

Any section of non-LIFO storage can be freed at any time; therefore a simple stacking mechanism cannot be used, because it would waste storage by leaving freed storage within the stack. A different method is therefore used. When storage that is contiguous with the major free area is freed, it is amalgamated with the major free area by altering the end-of-segment (EOS) pointer, which indicates the end of this area. When storage that is not contiguous with the major free area is freed, it is placed on the free-area chain, which is anchored to a field in the TCA appendage. Whenever an allocation is made, an attempt is made to place the allocation in an area that is already on the chain, rather than use a further section of the major free area. Allocations of non-LIFO dynamic storage are always handled by the library module IBMBPGR, whose address is held in the TCA. Figure 6.3 illustrates the principles involved. Whenever an allocation within the major free area is made, the end-of-segment (EOS) pointer, in the TCA, is updated to point to the end of the major free area.

If there is not sufficient space in either the major free area or on the free area chain, a GETMAIN macro instruction is issued for the required amount of storage. Non-LIFO storage acquired by a GETMAIN is freed by a FREEMAIN macro instruction.

ACQUIRING A NEW SEGMENT OF LIFO STORAGE

Every time a new procedure or block is entered, or a VDA is acquired, a test is made to see whether there is enough space, for the DSA or VDA, between the NAB pointer and the EOS pointer. If there is <u>not</u> enough space then an attempt is made to use the largest space on the free-area chain as a new segment for the DSA or VDA.

Pointers BOS and EOS in the TCA are set to point to the beginning and end, respectively, of the new segment. The DSA or VDA is allocated storage in the low-address end of the segment, and the NAB pointer is set to point to the first free byte after the DSA or VDA. The former values of BOS and EOS are stored at the start of the new segment.

A segment number is given to each segment, starting at hexadecimal "FF" and decreasing by 1 for each new segment. The number for the ISA is "FF", the second segment "FE", and so on. This number is held as the first byte of the NAB, BOS, and EOS pointers. The result of this device is that, when logical arithmetic is used, all addresses in later segments are apparently less than those in the earlier segments, regardless of their actual position. This simplifies segment handling. For instance, when a DSA in the second segment is freed, NAB is simply restored to its previous value which may well be in the first segment. NAB will then hold value "FF------", and EOS the value "FE------". When a further DSA is required, EOS will be less than the sum of NAB and the DSA length, as EOS is already less than NAB. Consequently it will appear that there is insufficient space for the DSA in the first segment, regardless of whether or not this is the case. The library module IBMBPGR is thus called to restore BOS and EOS, add the emptied segment to the free-area chain, and, if possible, place the new DSA after NAB in the first segment. The process is illustrated in figure 6.5.

IBMBPGR - STORAGE MANAGEMENT ROUTINE

The allocation and freeing of LIFO storage within a given segment is handled by compiled code or by the library module requiring the storage. All other dynamic storage allocation is carried out by the resident library routine IBMBPGR; this module has four entry points:

IBMBPGRA	Allocate non-LIFO storage.
IBMBPGRB	Free non-LIFO storage.
IBMBPGRC	Obtain and free additional storage segments (for DSAs).
IBMBPGRD	Obtaining and freeing additional storage segments (for VDAs).

These four entry points are described below. In all cases storage is allocated in multiples of 8 bytes.

Allocating Non-LIFO Storage (IBMBPGRA)

When entered by entry point IBMBPGRA, the module first searches the free-area chain, (if one exists) and allocates the storage in the smallest possible area on the chain. If there is no chain, or no area on the chain that is large enough, IBMBPGR attempts to allocate the storage in the area immediately preceding the EOS pointer. If there is not enough space between the EOS pointer and the current NAB pointer, a GETMAIN macro is issued for the required storage. If the GETMAIN cannot be satisfied, the system ends the job with an ABEND-code 80A. This ABEND is intercepted by the ABEND analyzer IBMBPES. IBMBPES puts out a message indicating which statement was being executed and when the demand for storage was made. It then returns to the system to complete the ABEND.

Provided that storage can be allocated, control is passed back, with register 1 pointing to the address of the storage allocated.

Freeing Non-LIFO Storage (IBMBPGRB)

When freeing non-LIFO storage or segments of LIFO storage IBMBPGR first tests to discover whether the element being freed is within the ISA. This is done by seeing if the address is between the value held in register 12, the address of the TCA, and the value held in the TISA field of the ISA which points to the end of the ISA. If the element is outside the ISA it must have been acquired with a GETMAIN macro instruction. It is therefore freed with a FREEMAIN macro instruction.

If the element to be freed is within the ISA, the module scans the free-area chain (if one exists) to see whether the storage being freed can be amalgamated with areas already on the chain. This is done if possible. The module then checks to see whether the storage being freed is adjacent to the major free area. If so, EOS is altered to point to the end of the area being freed or to the end of the amalgamated area, if this adjoins the major free area. If the element cannot be amalgamated with any other, the area is added to the free-area chain, which is arranged in descending order of addresses. The format of a free area chain element is shown in figure 6.4.

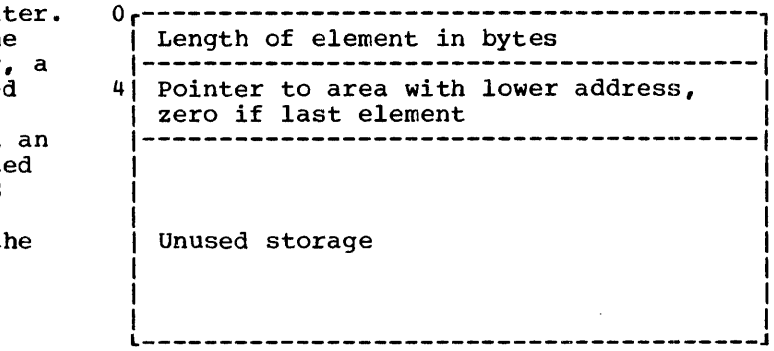


Figure 6.4. Format of element on free area chain

Segment Handling (IBMBPGRC and IBMBPGRD)

When compiled code discovers that the address contained in the pointer NAB plus the length of the new DSA or VDA to be allocated is greater than the value of the pointer EOS, IBMBPGR is called either at entry point C or entry point D depending on whether the storage is required for a DSA or for a VDA. Entry point C is used if a DSA is required, entry point D, if a VDA is required. The difference is the method used to store the caller's registers. IBMBPGRC stores the caller's registers in a special save area in the TCA, because no DSA has yet been acquired; IBMBPGRD stores the registers in the caller's DSA, in the usual manner.

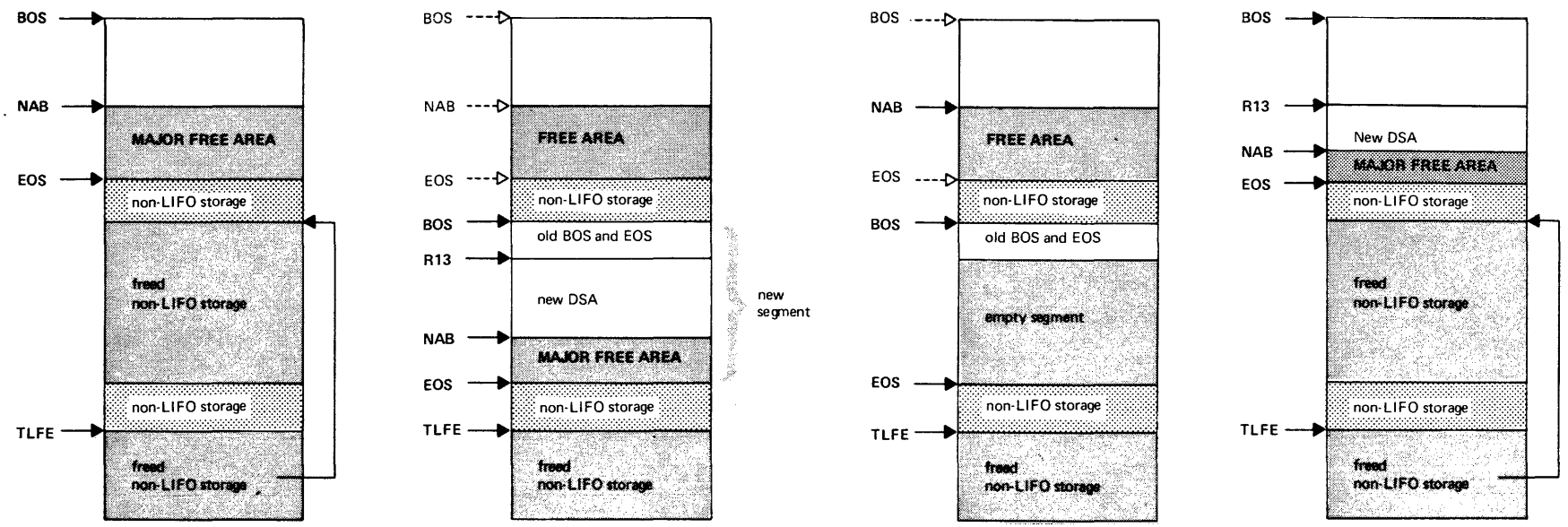


Figure 6.5. Principles involved in allocating and freeing segments of PL/I dynamic storage

The entry points are called in two circumstances:

1. There is insufficient room in the current segment for allocation of the DSA or VDA and, consequently, a new segment is required.

2. A segment other than the first one has been allocated, but is no longer in use.

IBMBPGRC and IBMBPGRD check to see which of the above two situations caused the call. This is done by determining whether the number in the first byte of NAB is greater than the number in the first byte of EOS.

In case 1 above, the segment numbers are the same, and a new segment must be allocated. A new segment is allocated by searching the free-area chain for the largest available area and using this as a new segment. If there is no area large enough to hold the new DSA, a GETMAIN macro instruction is issued and the new segment set up in the area acquired.

When a new segment is allocated, the old values of BOS and EOS are placed in control words at the head of the new segment. New values for BOS and EOS pointing to the beginning and end of the new segment, with first byte numbers decremented by one, are placed in the TCA. The address of the new NAB is passed in register zero; the address for the start of the new DSA or VDA is passed in register 1. The format of a secondary segment is shown in figure 6.6.

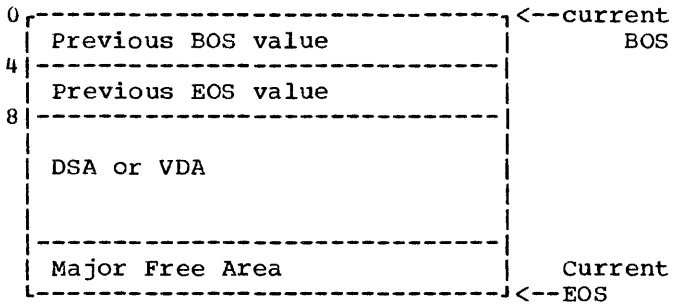


Figure 6.6. Format of second and subsequent segments of the LIFO stack

In case 2 above, the number in the first byte of NAB is greater than the number in the first byte of EOS. If the difference is greater than one, more than one extra segment has been allocated for DSAs or VDAs which are no longer current. In this case, segments are freed until only one empty segment remains. This is done by setting BOS and EOS to the values held in the control words at the head of each segment and freeing the storage in the way described for IBMBPGRB above.

When only one empty segment remains, a test is made to see whether the new DSA will fit into the segment that contains the present NAB pointer (the segment before the empty segment). This test is made by comparing the current NAB pointer with the old EOS pointer held in the control words of the empty segment. If there is sufficient room, the empty segment is freed as described under IBMBPGRB above. Return is then made to the caller, with a new value for EOS and BOS, and the DSA is allocated immediately after the old NAB.

If there is not enough room in the segment containing NAB, then a test is made to see if the empty segment is large enough to hold the new DSA. This is done by comparing the difference between the current BOS and EOS with the length of the element. If there is enough room, the DSA is allocated in the empty segment. The address of the start of the storage is passed to compiled code in general register 1, and the address of the new NAB passed in general register 0.

If there is not enough room in the empty segment, then the segment is freed. There are now no empty segments, and the situation is treated as if there had been no empty segments in the first place.

Note: It is possible that after freeing a number of empty segments, an area on the free-area chain can immediately follow EOS. However, the possibility is remote, and no check is made to see whether this is the case.

Storage Reports

When the PL/I programmer requests a storage report, he is given, after the completion of his program, a report showing the following:

1. The ISA size specified (if a size was specified).

2. The ISA size used.

3. The amount of PL/I storage required by the program. (This is a suggested optimum ISA size.)

4. The maximum amount of storage obtained outside the ISA at any one time.

5. The number of GETMAIN macro instructions issued.

6. The number of FREEMAIN macro instructions issued.

7. The number of requests to acquire non-LIFO storage.

8. The number of requests to free non-LIFO storage.

The report is generated by the storage report routine, IBMBPGD. This module is loaded during program initialization, instead of the normal storage management module IBMBPGR. IBMBPGD has the same entry points and carries out the same functions as IBMBPGR. However, it also maintains a record of certain storage statistics. To ensure that IBMBPGD handles all storage allocation both inside and outside the ISA, the EOS field in the TCA is set with a dummy value so that the storage routine will be called whenever LIFO storage is required, as well as for non-LIFO storage and stack overflow requests.

The storage report is issued during program termination. The termination routine, IBMBPIT, calls the report writing module, IBMBPMR. The report is transmitted to the dump file.

Action during Initialization

During program initialization, if REPORT has been included in the parameters passed to IBMBPIR, the report storage management routine IBMBPGD is loaded, and its entry point addresses placed in the TCA. The value in the end-of-segment pointer, EOS, is then set to zero. Space for a report table is acquired, and the true value of the end of segment placed in a field in the report table.

Action during Execution

During execution, IBMBPGD is called every time there is a request for PL/I dynamic storage. It is called for non-LIFO storage in the normal way, and, when LIFO storage is required, it is called because the zero value in EOS results in the value of NAB+DSA or VDA being greater than EOS. Consequently, the stack overflow routine (IBMBPGD, entry point C or D) is called. When a call is made to entry points C or D, IBMBPGD makes a test against the true value of the end of segment held in the report table, and, if there is sufficient room, the storage is acquired in the current segment of the LIFO stack. If there is not sufficient room, IBMBPGD takes the same action as IBMBPGRA (described earlier in this chapter).

All other storage acquisition by IBMBPGD is handled in exactly the same way as for the corresponding entry point of IBMBPGR. However, IBMBPGD keeps a running total of the following in the storage report table.

1. The highest value obtained by subtracting the current length of the major free area from the current amount of PL/I storage acquired outside the ISA.

2. The largest amount of PL/I storage obtained outside the ISA at any one time.

3. The number of GETMAIN macro instructions issued.

4. The number of FREEMAIN macro instructions issued.

5. The number of requests to acquire non-LIFO storage.

6. The number of requests to free non-LIFO storage.

The values are altered if necessary every time IBMBPGD is entered. The value of (1) and (2) above is calculated on every call, and the highest number retained in the report table. The format of the storage report table is given in appendix A.

Action on Termination

On termination, the termination routine, IBMBPIT, calls the storage report writing module, IBMBPMR, which transmits the storage report onto the dump file.

The amount of PL/I storage required is calculated by adding the figure described in (1) above to the ISA size used. The figure will be positive if any storage outside the ISA was acquired; it will be negative or zero if no storage was acquired outside the ISA.

Two things should be noted about the results produced by a storage report.

1. If storage was acquired outside the ISA, the figure given for storage used cannot be taken as final. A further request for a report when the program is run in the ISA size suggested may result in a smaller figure being generated. This smaller size should be used. This discrepancy is caused by the differences in acquiring

storage inside and outside the ISA. To obtain a correct figure using only one run, the program should be run in a large ISA that can be expected to hold all PL/I storage.

2. The report can only refer to the particular run of the program on which the report was given. Runs with different data or parameters may have different storage requirements.

The modules IBMBPGD, IBMBPMR, and the initialization and termination modules are fully described in PL/I transient library program logic manual.

Storage Reports for Multitasking Programs

Storage reports for multitasking programs are generated in the same way as those for non-multitasking programs. A special storage management module is loaded at execution time, and this retains statistics of the amount of storage used. To ensure this module handles all requests for storage, the value in EOS is set to zero, and the true EOS value is retained in the report table. The report is issued during program termination by the module IBMBPMR.

For a multitasking storage report the following information is given:

For the major task:

The same as for a non-multitasking program (see above).

For subtasks, a combined report for all subtasks showing:

The maximum ISA size used by any subtask

The minimum ISA size used by any subtask

The maximum PL/I storage required by any subtask

The minimum PL/I storage required by any subtask

The maximum amount of storage acquired outside the ISA by any subtask

The minimum amount of storage acquired outside the ISA by any subtask

The total number of GETMAIN and FREEMAIN macro instructions issued by all subtasks

The total number of requests to free and acquire non-LIFO storage issued by all tasks

To enable these figures to be produced, a multitasking version of the storage report module is used. This module, IBMTPGD, has two more entry points than its non-multitasking counterpart. These are:

IBMTPGDE - called when a task is initialized.

IBMTPGDF - called when a task is terminated.

IBMTPGDE is called when a task is initialized. It acquires storage for the report table for the task, and retains a record of the number of active PL/I tasks, increasing the maximum number if necessary.

IBMBPGDF is called when a task is terminated. If the terminated task is a subtask, IBMBPGDF completes the relevant field in the subtask storage report table, from information in the report table of the terminating task.

During initialization, space is required by the control task for a combined subtask report table which will hold the information from which the merged subtask report will be generated. During the initialization of each task, space for a report table for that task is obtained. The report table for the major task is flagged.

Throughout the execution of each task, a separate report table is maintained. At the end of each subtask, the information in the terminating task is merged into the combined subtask table, held in the storage associated with the control task.

When the jobstep is terminated, IBMBPMR produces the information from the merged subtask report table and the report table of the major task. (IBMBPMR is used to output the report for both tasking and non-tasking programs.)

Storage Management in Programmer-allocated Areas

By using area variables, the programmer can obtain a continuous area of storage for based variables. The allocation of storage for area variables is handled in the same way as that for other types of variable, and depends on the variable's storage class. The allocation and freeing of storage within an area is handled by the library module IBMBPAM.

IBMBPAM keeps a check on the amount of

storage allocated. If there is not enough space for an allocation, or if the target area is too small to hold the source area assignment statement, the AREA condition is raised.

The method employed is that storage is allocated from the low-address end of the area, and an offset is kept to the end of the item with the highest address in the area. This offset is known as OEE (offset to end of extent). When storage is freed, either the OEE is altered or the storage is placed on a free-storage chain, with the largest segment at the start of the chain.

Before a space is freed, a check is made to see whether it is contiguous with a space or spaces that are already on the free storage chain. If it is, the contiguous spaces are amalgamated. A check is then made to see whether the amalgamated space is contiguous with the OEE. If the space is contiguous with the OEE, the OEE is pointed to the start of the space, and the space removed from the free storage chain. If the amalgamated space is not contiguous with the OEE, the free area chain is rearranged so that it is in the correct order.

If the space to be freed is not contiguous with another space on the free storage chain, a check is made to see if it is contiguous with the OEE. If it is, the OEE is updated.

If the space to be freed is contiguous neither with the OEE nor with another space on the free storage chain, the space is placed in its correct position in the storage chain.

When a free chain exists, IBMBPAM always attempts to allocate storage by using a space on the chain. The low-address end of the smallest possible space on the chain is used, and the chain is then rearranged to maintain the correct order of decreasing size.

Multitasking Considerations

Storage handling within each task follows the pattern described above, except that certain storage requests are made for storage that will be available to all tasks. This storage has to be obtained in subpool 0. To indicate such a requirement, IBMTPGR is called with a negative value. A GETMAIN for the specified amount is then issued to subpool 0, the negative value indicating what the storage must be in subpool 0.

The method used to acquire the ISA is slightly different for tasking. This is described below.

Acquiring the ISA when Multitasking

The size of the ISA required for the major task and every minor task can be requested in the ISASIZE parameter of the EXEC card. If the size in the parameter is smaller than that needed for the program management area, only the exact size required for the program management area is acquired and all further allocations of dynamic storage are made by issuing GETMAIN macro instructions. These allocations are made in exactly the same way as they are when non-tasking programs cannot acquire space within the ISA, see above under "IBMBPGR - Program Management Routine".

The default action, taken if no ISA size is specified, is to acquire storage for all ISAs in multiples of 4K bytes. If the program management area can be contained in 4K bytes (which will normally be the case) only 4K bytes are acquired and this is set up as the ISA. If the program management area contains more than 4K bytes, (an exceptionally large PRV might cause this) a further 4K bytes are acquired. This process continues until enough space is acquired for the program management area.

4K bytes of storage will normally be enough to hold the program management area and the DSA for the main procedure.

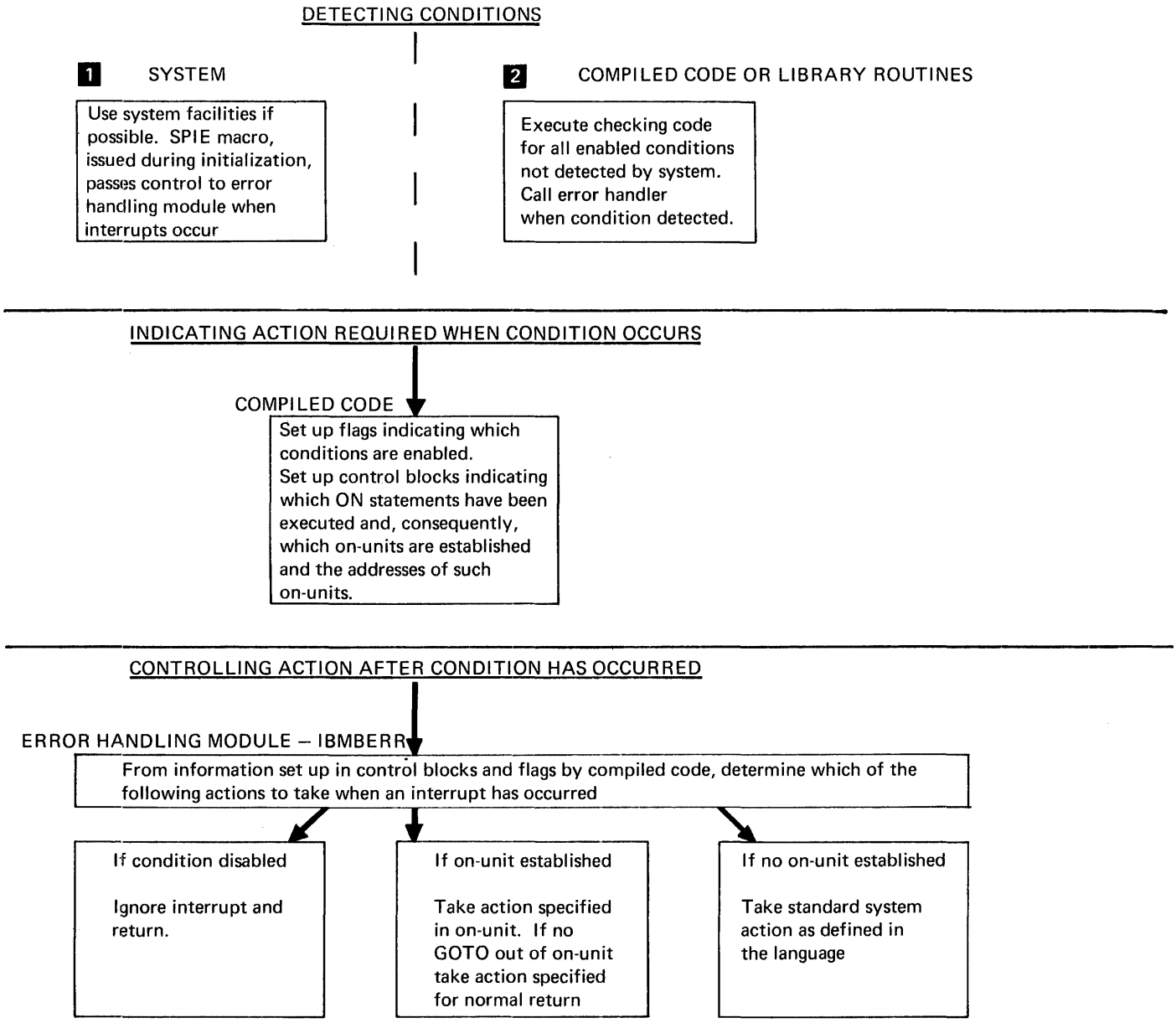


Figure 7.1. The principles of error handling

Chapter 7: Error and Condition Handling

This chapter deals with the method used to implement execution time error handling. All errors detected at execution time are associated with PL/I conditions and can be handled either by on-units written by the programmer or by standard system action, as defined in the PL/I language.

The chapter starts with a brief discussion of the terms and concepts used in error handling. A discussion of the error handling facilities offered by the operating system and those specified in the PL/I language follows. The implementation problems these facilities raise and the method used to solve them are then described. A separate section is devoted to the CHECK condition because this raises special problems. The chapter is completed by a brief discussion of the error message modules, the modules used to implement the PLIDUMP facility, and the handling of the compiler FLOW option.

Error detection during compilation is not covered in this chapter. Nor is any advice given on how to use PL/I error handling facilities. Advice on debugging with dumps is given in chapter 12.

Note: If the NOSPIE or NOSTAE options are specified in the parameters for the procedure, much of what is said in this chapter does not apply. The PL/I SPIE or STAE macros will not be issued and system detected interrupts and ABENDs will not be handled in the PL/I defined manner.

Terminology

Throughout this chapter a number of special terms are used. Some of them are terms used in the PL/I language, others are terms that are used to describe certain implementation features and concepts. The terms are listed below.

Established: This term is used to describe on-units and, sometimes, ON statements. The on-unit or statement is said to be "established", if the action specified in the on-unit or ON-statement will be taken should the specified condition arise. Thus an on-unit becomes established when the ON statement is executed and ceases to be established when an ON or REVERT statement referring to the same condition is executed, or when the associated block is terminated.

Enabled: This term is used to describe certain PL/I conditions (SIZE, CONVERSION, etc.). A condition is enabled when the occurrence of the condition will result in the execution of an on-unit or standard action. A condition is disabled when the occurrence of the condition will, apparently, be ignored.

Qualified and Unqualified Conditions: Qualified conditions are those conditions, such as ENDPAGE, that need to be qualified by a file or other name. Unqualified conditions are those that do not need qualification. Figure 7.3 shows which conditions are qualified and which are unqualified.

Program Check and Software Interrupts: Certain PL/I conditions are detected automatically by the computing system. Others have to be detected by special checking code either in library modules or in the compiled program. Interrupts detected by the system are referred to as program check. Interrupts detected by special checking code are referred to as software-detected or software interrupts. A list of program check interrupts and their associated PL/I conditions is given in figure 7.2.

These terms program check and software interrupts are used for convenience in this publication and are not accepted terms in the PL/I language. Figure 7.3 shows which interrupts are system detected and which are software detected.

Static and Dynamic Descendency: Static and dynamic descendency are terms used to define the scope of PL/I features. On-units are dynamically descendent. That is, they are inherited from the calling procedure in all circumstances. Condition enablement is statically descendent. That is, it is inherited from the containing block in the source program. Static descendency can be determined during compilation. Dynamic descendency cannot be known until execution. See figure 7.4.

Normal Return: Normal return is return from a called block by means of reaching the END or RETURN statement rather than because of a GOTO out of the block. In an error-handling context, normal return is taken to mean normal return from the on-unit. The action taken after normal return from an on-unit is specified in the PL/I language. For most conditions, it is to return to the point of interrupt.

Standard System Action: Standard system action is the name given to the default PL/I-defined action taken when a condition occurs and there is no established on-unit for that condition.

Machine interrupt	PL/I condition
Operation Privileged operation Execute Protection Addressing Specification Data	ERROR (after issuing a message)
Fixed-point overflow	FIXEDOVERFLOW/SIZE
Fixed-point divide	ZERODIVIDE/SIZE
Decimal overflow	FIXEDOVERFLOW/SIZE
Exponent overflow	OVERFLOW
Exponent underflow	UNDERFLOW
Floating-point divide	ZERODIVIDE

Figure 7.2. Machine interrupts associated with PL/I conditions

Background to Error Handling

System Facilities

The operating system offers certain error-handling facilities. These can be summarized as follows:

Various situations can cause an machine interrupt which results in entry to the supervisor. It is possible for the programmer to define the action that will be taken after any of these interrupts by means of a routine specified in a SPIE macro instruction. Alternatively, the programmer can accept the default action of the system. It is also possible for the programmer to prevent the occurrence of certain interrupts by masking out fields in the PSW.

PL/I FACILITIES

The PL/I language offers similar but greatly extended facilities. The number of situations causing interrupts is considerably larger and some, such as ENDFILE, can be used to control normal program flow rather than to handle errors. The use of on-units allows the programmer to obtain control after any interrupt. Alternatively he can accept standard system action. The programmer also has the choice of whether certain of the conditions will cause interrupts. This is done by enabling or disabling the conditions. If the condition is disabled neither an unit ncr standard system action will be taken if the condition occurs.

A number of PL/I conditions correspond directly to the interrupts that are detected by the operating system (see figure 7.2). Other conditions however belong only to PL/I.

The majority of PL/I conditions are caused by errors in program logic or the data supplied. Some, however, are not connected with errors. These are conditions such as ENDFILE, which occur at unpredictable times and consequently cannct easily be anticipated by code in the source program.

Conditions that are most probably caused by programming errors are known as error conditions. Figure 7.3 shows which conditions are error conditions. The standard system action for these conditions is to put out a message and raise the ERROR condition.

The ERROR condition is also raised by any programming error that is not directly covered by a PL/I condition. A data interrupt, for example, raises the ERROR condition, and certain software detected conditions, such as taking the square root of a real negative number, also raise the ERROR condition.

The ERROR condition consequently gives the programmer blanket coverage of all program errors. The ERROR condition differs from other conditions in that a diagnostic message is always generated regardless of whether an ERROR on-unit exists. If an on-unit exists, the message is generated before on-unit action is taken.

A further facility offered by PL/I is the availability of condition built-in functions and pseudo-variables. These allow the programmer to inspect various fields associated with the interrupt and, in certain cases, to alter the contents of these fields.

The situation in PL/I is complicated by the question of the scope of on-units and condition enablement. Condition enablement is statically descendent and can be decided during compilation. On-units, however, are dynamically descendent and the establishment or otherwise of on-units can only be decided during execution. (See "Terminology" above.)

Name of condition	Qualified	Description	Recognized by	Default	Programmer Control	ERROR** Condition
Computational						
CONVERSION	no	Attempt to convert invalid character string	Code in relevant library modules	enabled	yes	yes
FIXEDOVERFLOW	no	Overflow of a fixed point value	System	enabled	yes	yes
SIZE	no	Attempt to assign too large a value	Compiler-generated checking code, or hardware	disabled	yes	yes
OVERFLOW	no	Overflow of a floating-point value	System	enabled	yes	yes
UNDERFLOW	no	Exponent becomes smaller than permitted minimum	System	enabled	yes	no
ZERODIVIDE	no	Attempt to divide by zero	System	enabled	yes	yes
Input/Output						
ENDFILE	yes	End of file reached	Code in relevant library modules	enabled	no	yes
ENDPAGE	yes	End of a page on a print file reached	Code in relevant library modules	enabled	no	no
TRANSMIT	yes	Transmission error on a file	Code in library modules	enabled	no	yes
UNDEFINEDFILE	yes	Error in opening file	Code in relevant library modules	enabled	no	yes
KEY	yes	Invalid key	Code in relevant library modules	enabled	no	yes
NAME	yes	Unrecognizable data-directed input	Code in relevant library modules	enabled	no	no
RECORD	yes	Incorrect size record	Code in relevant library modules	enabled	no	yes

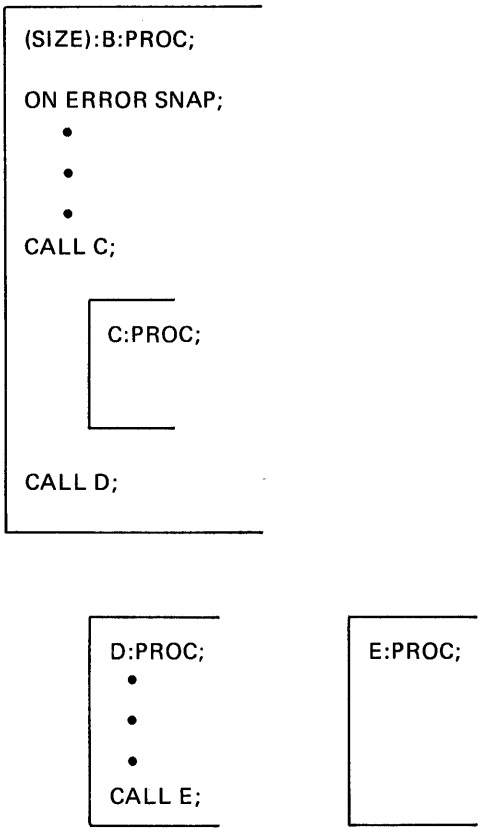
Figure 7.3. (Part 1 of 2). PL/I conditions

Name of Condition	Qualified	Description	Recognized by	Default	Programmer control	ERROR** condition
Program Checkout						
SUBSCRIPTRANGE	no	Array subscript outside declared bounds	Compiler-generated checking code	disabled	yes	yes
STRINGSIZE	no	Attempt to assign a string of too great length	Code in relevant library modules	disabled	yes	nc
STRINGRANGE	no	Attempt to access beyond limits of string	Code in relevant* library modules	disabled	yes	no
CHECK (variable or label)	yes/no	Value assigned to identifier or control passed through label	Compiler-generated checking code, or library module	disabled	yes	no
List Processing						
AREA	no	Attempt to allocate beyond end of area	Relevant library modules	enabled	no	yes
System Action						
ERROR	no	Any error condition including those not covered by other conditions**	Relevant library modules, or compiled code, or system	enabled	no	-
FINISH	no	Program about to be terminated	Relevant library modules	enabled	no	-
Programmer Named						
CONDITION (name)	no	Programmer defined condition	Signal statement	enabled (when coded)	no	-

* When STRINGRANGE is enabled, library modules are always called to handle substring operations. These modules have the necessary code for checking for the STRINGRANGE condition.

** The ERROR condition is raised when an error occurs that is not covered by PL/I exceptional conditions - taking the square root of a real negative number, for example. It is also raised as standard system action when handling all types of error conditions. Thus an ERROR on-unit enables the programmer to intercept all error conditions.

Figure 7.3. (Part 2 of 2). PL/I conditions



Static descendency: the enablement prefix (SIZE): in procedure B is inherited only by the contained procedure C, regardless of which procedure calls which.

Dynamic descendency: the on-unit ON ERROR SNAP; is inherited by any procedure called by B and any subsequently called procedures. Thus, if B calls D, which calls E, the on-unit is established in procedure E.

Figure 7.4. Static and dynamic descendency

UNQUALIFIED CONDITIONS

1. A flag at the head of the DSA indicates that static ONCBs exist for that block.
2. The block and current enable cells indicate which of those conditions that are under programmer control are enabled at any given point in the program. Each such condition is represented by a single bit in each cell.
3. There is an on-cell for every ON-statement in the block. Each on-cell consists of a one-byte code identifying the condition, e.g., X'0A' (SUBSCRIPTRANGE). If the same condition appears more than once, previous on-cells are set to zero.
4. Static ONCBs are held contiguously in static storage, in the same order as the corresponding on-cells. They contain a code byte and flags that indicate such things as : whether SYSTEM was specified, whether SNAP was specified, whether the on-unit consists of a single GOTO statement, whether it is a null on-unit, etc. If there is an on-unit, its address is given in the second byte. (For GOTO-only on-units, the offset of the address of the label variable is given.)

QUALIFIED CONDITIONS

1. A flag at the head of the DSA indicates that dynamic ONCBs exist.
2. Dynamic ONCBs are set up during execution of each block in which qualified condition ON-statements occur. The last two words of a dynamic ONCB contain the same type of information as static ONCBs (described above, under 'Unqualified Conditions'), but use additional flags to indicate whether the condition is enabled and whether it is established. The second word contains qualifying information, such as the address of the FCB (for conditions such as ENDFILE, RECORD, TRANSMIT, KEY, etc.), or address of a symbol table (for ON CHECK on-units).
3. Dynamic ONCBs are chained together, the most recent being addressed from a fixed offset in the DSA. The last dynamic ONCB in the chain contains zero in its backchain field.

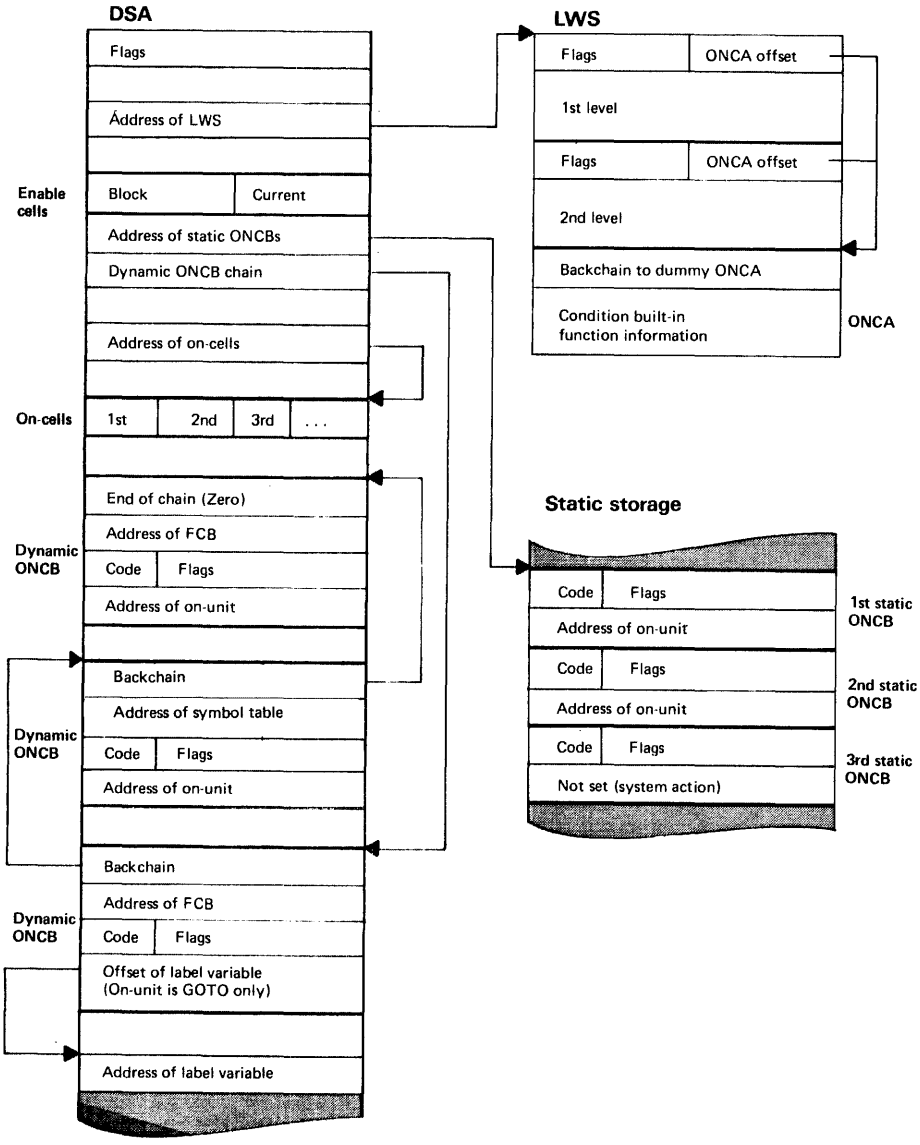


Figure 7.5. The major fields used in error handling

Implementation of Error Handling

To implement the PL/I error handling scheme it is necessary to be able to detect all the PL/I conditions, to acquire various information about how the conditions occurred for condition built-in function values, to determine whether the condition is enabled and whether an on-unit is established, and then take the necessary action.

The methods used by the PL/I optimizing compiler are summarized below.

1. Detection of the PL/I conditions

   All PL/I conditions that correspond directly to program check interrupts are left to the detection of the operating system.

   A SPIE macro, issued during program initialization, results in control being passed to the error handling module IBMBERR.

   All other interrupts are detected by special checking-code, either generated by the compiler, or included in library modules. The checking-code calls the error handling module IBMBERR when a condition is detected.

2. Acquiring information about the interrupt

   Information about the interrupt is obtained by analyzing the PSW for program check interrupts and by checking-code for software detected interrupts. Condition built-in function values are accessed through a control block known as the ON communications area(ONCA).

   For software detected conditions, the ONCA is largely set up by the checking-code. For system detected conditions the ONCA is set up by the error handler from information in the PSW.

3. Compilation and handling of on-units

   Certain simple on-units are represented by a series of flags in an ON control block (ONCB), but the majority are compiled as independent program blocks to which control is passed from the error handling module.

4. Maintaining a record of enablement and establishment

   During execution, information indicating which conditions are enabled and which on-units are established is placed in the following control blocks:

   Enable cells - indicating enablement or disablement of the conditions that can be enabled and disabled by the programmer.

   ON cells - indicating which unqualified conditions have established on-units.

   ON control blocks (ONCBs) - indicating address of on-units or action to be taken, and, for qualified conditions, whether the on-unit is established, and, for CHECK only whether the condition is enabled.

5. Determining and directing action when interrupt occurs

   After every interrupt, control is passed to the error-handling module IBMBERR.

   A test is first made to see whether the condition is one that may be enabled or disabled by the programmer. If the condition is disabled, control is returned to the point of interrupt. If the condition is enabled, a search is made in all active blocks for an established on-unit. This is done by examining ON cells or ONCBs set up by compiled code. If an on-unit is found, the specified action is taken. If the dummy DSA is reached without finding an on-unit, standard system action is taken under the control of the error-handling module.

The scheme is shown diagramatically in figure 7.1, and each topic is discussed in greater detail in the sections below. A summary of the uses of the various control blocks is given in figure 7.5.

Figure 7.6 gives a programming example in which the error handling actions can be followed through. Figure 7.15 summarizes the complete error handling operation. It is intended for reference throughout the chapter and for use as a reminder by readers who know the basic principles.

The handling of the CHECK condition, which is a special case, is treated in a separate section of this chapter under the heading "The CHECK Condition."

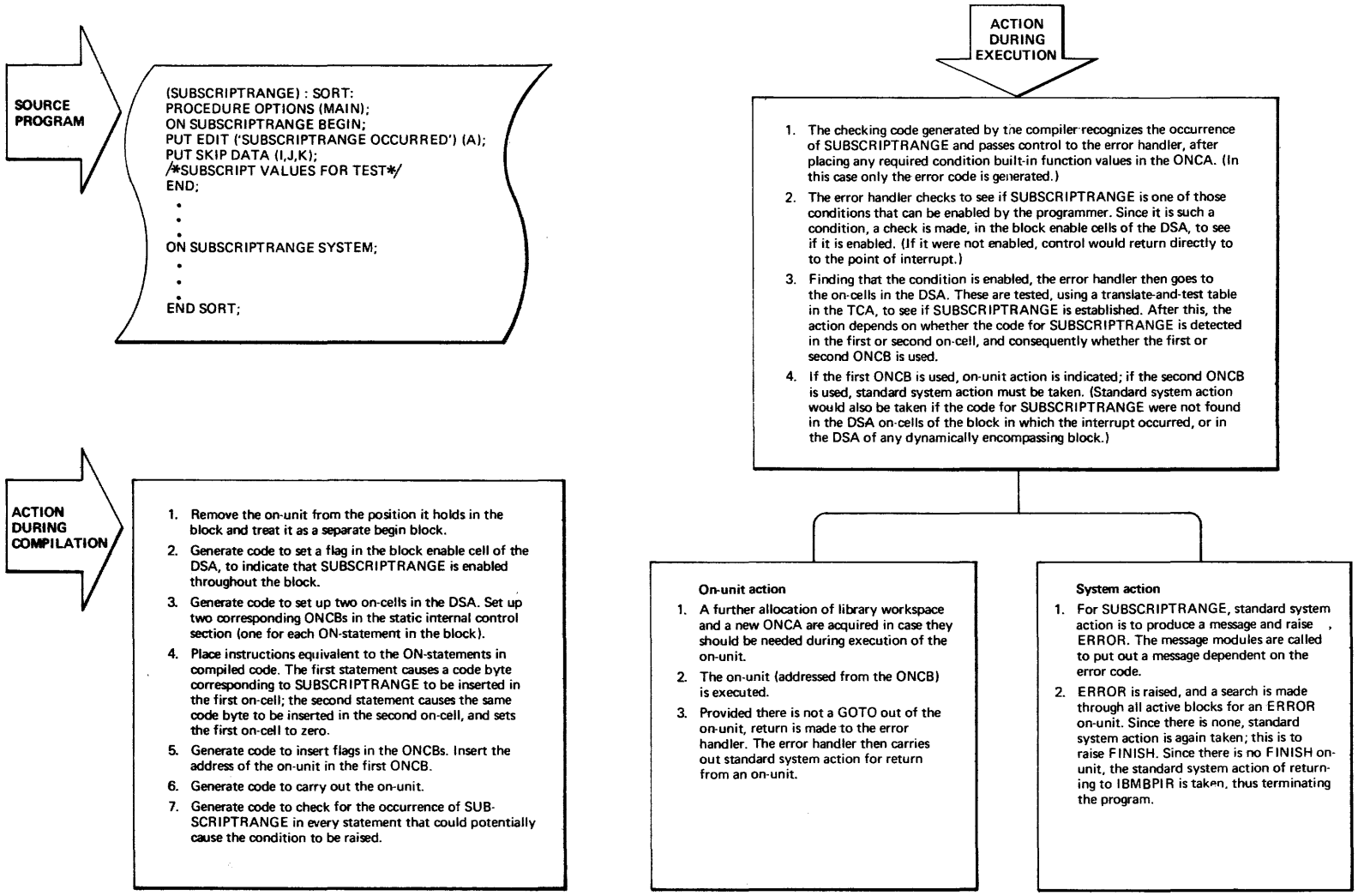


Figure 7.6. An example of error handling

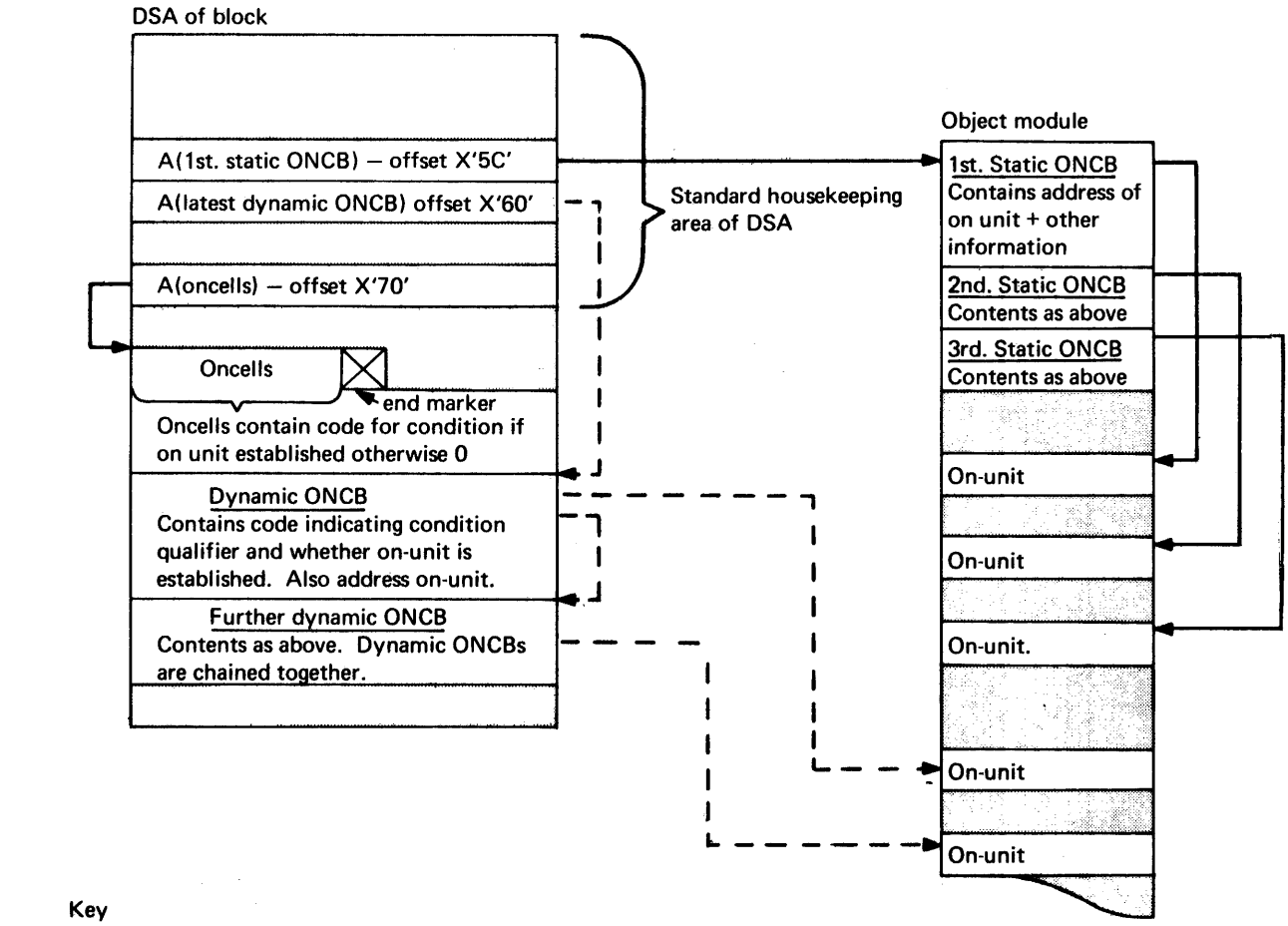


Figure 7.7. Addressing on-units

Detecting the Occurence of Conditions

SYSTEM DETECTED CONDITIONS

As far as possible, the detection of PL/I conditions is left in the hands of the operating system. Those conditions that can be detected by the operating system are left in the hands of the operating system. The only interrupt that is masked out in the PSW is the significance interrupt. Regardless of the enablement or disablement of PL/I conditions no other interrupts are inhibited.

When a condition is detected by the system, a SPIE macro, executed during program initialization, causes control to be passed to entry point A of the error-handling module IBMBERR. The address of this point is held in the TCA appendage. When entered by this entry point the error handler equates the interrupt with a PL/I condition and passes control to the main error handling logic of the module. The relationship between PL/I conditions and system interrupts is shown in figure 7.2.

SOFTWARE DETECTED CONDITIONS

During compilation, the compiler analyzes the conditions enabled for each block and statement and ensures that the necessary checking code will be executed. The checking code may be specially generated by the compiler, or it may be included in library modules that will be called when the particular condition is enabled. The method used for checking for each condition is shown in figure 7.3.

As far as possible the checking code is not included in the program if the condition that it checks for is not enabled. However, every library module contains the checking code for detecting any PL/I condition that can occur in the module. In certain circumstances, therefore, code to check for software detected conditions will be executed and a call made to the error handler even though the condition is disabled.

When an interrupt has been detected during execution, the checking code sets up a parameter list for the error handling module IBMBERR. This parameter list, known as the interrupt control block, contains a code that defines the type of interrupt that has occurred and, if the condition is qualified, contains a means of identifying the qualifier. The checking code also calculates the value of relevant built-in functions and places these values, or their addresses in a control block known as the ON communications area (ONCA).

When these actions have been carried out a call is made to entry point B of the error handling module IBMBERR. The address of this entry point is held at offset X'78' in the TCA.

Detecting I/O Conditions

The TRANSMIT and the ENDFILE condition are normally detected by the data management routines rather than by PL/I code. When this occurs the error or end-of-file routine in the PL/I transmitter modules receives control and passes it to the error handler via a special I/O error module. This I/O error module contains the necessary code to set up the interrupt control block, including the error code and the qualifier. These conditions can, therefore, be considered to be software detected. Further detail is given in chapter 8 - Record Oriented Input/Output.

EXECUTING SIGNAL STATEMENTS

SIGNAL statements take the same form as software detected interrupts, they are executed by a call to IBMBERR with the appropriate interrupt control block. The error code in the interrupt control block will indicate, to the error handler, the type of condition signalled, and the fact that the condition was signalled. The call to the error handler is made to entry point B, regardless of whether the condition is normally detected by system or software.

It is necessary for the error handler to know that the condition was signalled, because different action may be required if the interrupt was signalled when computing certain built-in function values.

PASSING INFORMATION ABOUT INTERRUPT

When the error handling is entered it must be able to access information about the interrupt. This information must identify the type of condition that has occurred and further identify the interrupt so that the most useful diagnostic message can be generated. Any relevant built-in function values must also be available, plus the default values for any built-in functions that are not relevant to the type of

interrupt.

When the interrupt is software detected, some of the information is set up in the checking code before control is passed to the error handler. When the interrupt is system detected, the PSW is used and the error handler interprets the information in the PSW, setting up information in a format similar to that produced by the checking code. This allows the main logic of the error handler to treat program checks and software detected condition in the same manner.

The parameters passed to the error handler by compiled code are known as the interrupt block, and take the following format:

Word 1 Error code

Word 2 Qualifier if any

Words 3,4 and 5
extra information used in handling CHECK

The error code defines the type of error. The qualifier gives a method of identifying the qualifier for qualified conditions. For I/O conditions the address of the DCLCB, is used as a qualifier. The address of a symbol table, control section, or pseudo register offset is used for other qualified conditions.

The address of software detected interrupt is taken from the register 14 value when the error handler is called with a BALR 14, 15. This value is stored in the DSA by the prologue of the error handler. When the interrupt is system detected the address is taken from the PSW.

Error Code

The error code is either a two or four byte code that defines the reason for the interrupt. For all conditions except the error condition a four byte code is passed. For the errors that will immediately raise the ERROR condition only a two byte code is passed.

The four byte code is made up as follows:

Byte 1 identifies PL/I condition

Byte 2 identifies causes of condition

Byte 3 & 4 identify those ON built-in functions that are valid for the condition.

The two byte error code is raised only for the ERROR condition. The ERROR condition is raised for those interrupts and errors that have no directly associated PL/I condition. Certain of these, such as taking the square root of a real negative number, are software detected. Others are associated with program check interrupts such as a data interrupt.

When the error condition is to be raised a two byte code only is generated. The value in this code corresponds with a table held in the error handler which identifies the cause of the interrupt.

Condition Built-in Functions

Certain condition built-in function values are implicit in the information that is passed to the error handler. ONCODE, for example, bears a direct relationship to the error code. Other values, such as ONCHAR and ONSOURCE must be calculated when the interrupt occurs. These values or the addresses of the values are placed in the ONCA. The ONCA is addressed from library workspace. The address of library workspace is held at a fixed offset in every DSA. ONCODE, ONLOC, and ONFILE are not generated by the checking code as their contents are implicit in the information passed to the error handler.

The ONCODE is deduced from the error code and, when required, a transient library module IBMBEOC is called to translate the error code into the ONCODE. Both an error code and an ONCODE are used as it is possible to define the error more accurately than can be done with the ONCODES, which must be kept compatible with other PL/I compilers. Thus the error code allows a more useful diagnostic message to be generated than would be possible if only the ONCODE was generated.

The ONLOC value is also calculated by a separate module. ONFILE is accessed from the DCLCB. Both ONLOC and ONFILE are placed in the ONCA only if an on-unit is to be entered. Similarly if an on-unit is to be entered the error code is placed in the ONCODE field of the DSA. If the ONCODE value is required in the on-unit the module IBMBEOC is called to calculate the ONCODE from the error code.

Chain of ONCAs

PL/I allows access to condition built-in function values when no condition has

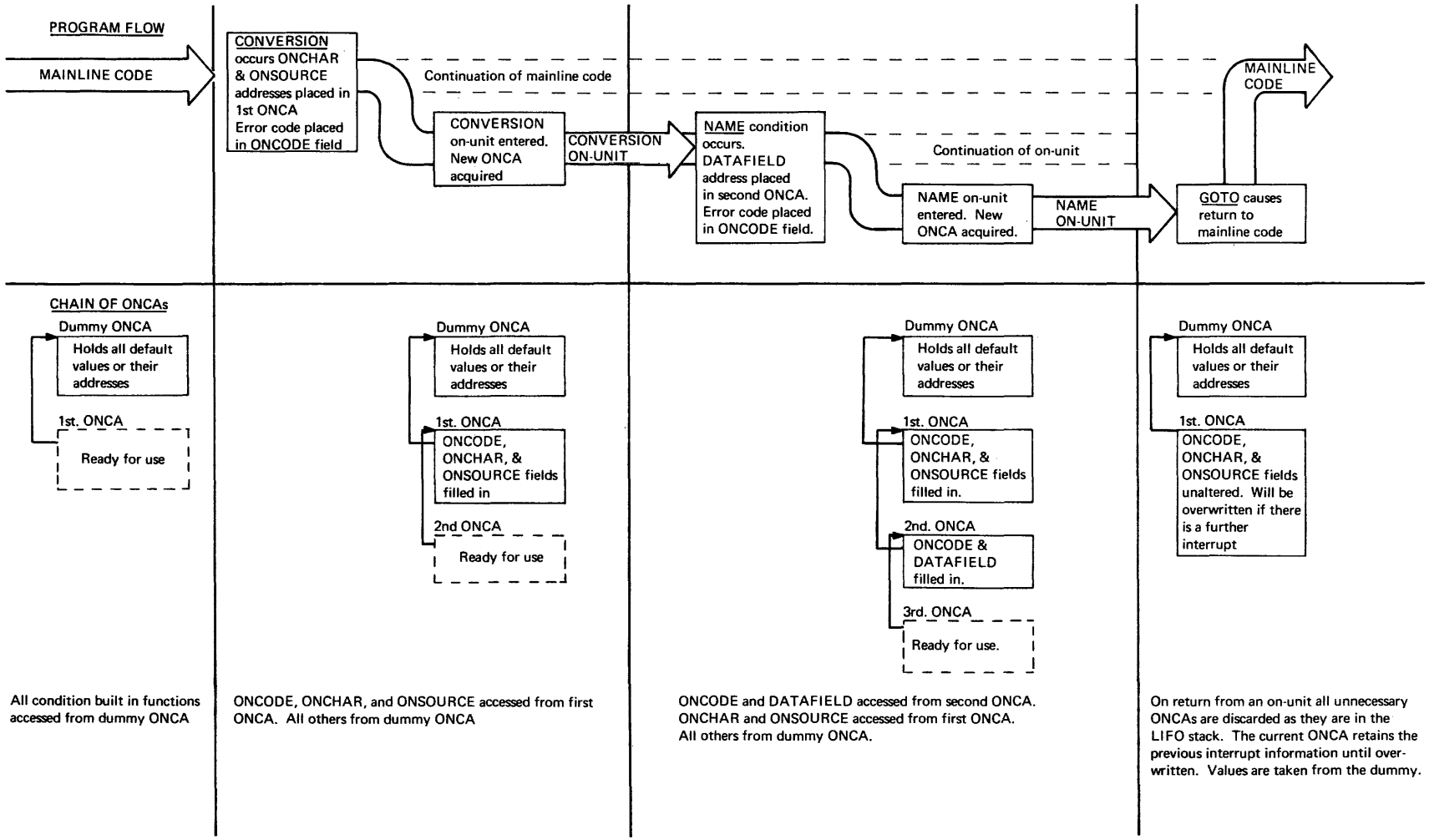


Figure 7.8. Accessing a built-in function value from the chain of ONCAs

occurred or when a condition has occurred in which the built-in function is invalid. The rule is, that the built-in function value given will be the most recent value in an active ONCA or the default value. To allow for this, ONCAs are chained together and the end of the chain is the dummy ONCA that is set up in the program management area during program initialization. The dummy has the same format as other ONCAs and contains the default values or pointers to the default values for all built-in functions.

For every interrupt that occurs, a new ONCA is acquired. This means that, should a condition occur within an on-unit, an ONCA will be available in which to place any relevant built-in function values or their addresses. A new allocation of library workspace(LWS) is also required for use during the on-unit.

When a built-in function value is required, the ONCA before the current ONCA is inspected. The current ONCA is unused as it is ready for a new set of values. Each ONCA is headed by flags that indicate which built-in functions are given in the ONCA. When the required built-in function value is flagged as invalid, a chain back is made to the previous ONCA. As all fields are valid in the dummy, the default will be used if there have been no interrupts for which the function is valid.

In the program below, an example of the chain of ONCAs is shown. The ONCHAR reference in the NAME on-unit would be valid if the NAME condition was raised in the CONVERSION on unit. The correct value would be accessed after chaining back to the ONCA associated with the CONVERSION interrupt.

In other circumstances the default value would be accessed from the dummy ONCA.

```
CHAIN:  PROC OPTIONS(MAIN);

        ON NAME BEGIN;    /*NAME ON
        UNIT*/
        PUT DATA(ONCHAR);
        .
        GOTO LABEL1;
        END;

        ON CONVERSION BEGIN;
        /*CONVERSION ON UNIT*/
        .
        GET DATA (A,B,C);
        .
        .
        END;

LABEL1:  X=Y+2;
        .
        END CHAIN;
```

A situation that could occur in this program, and the associated chaining of ONCAs are shown in figure 7.8.

When an on-unit is completed, the latest generation of LWS and the ONCA are deleted immediately control returns to a block before the error handler. This is because they are held as VDAs associated with the error handler's DSA. When control leaves the error handler, the current ONCA will contain the interrupt information for the original interrupt. This information remains until the ONCA is freed or a further interrupt occurs, in which case it is overwritten. (See figure 7.8.)

Establishment and Enablement Information

(Executing ON Statements)

Establishment and enablement information is set up and updated by compiled code. Enablement is indicated by a set of flags known as the "current enable cells," which are held in every compiled code DSA. Establishment for unqualified conditions is indicated by a further series of bytes in the DSA known as the "ON-cells." Establishment for qualified conditions is indicated in flags in dynamic ONCBs. Dynamic ONCBs are held in the DSA of the block in which the associated ON-statement occurs.

To alter the enablement for the duration of a statement or to execute an ON statement, compiled code alters the appropriate fields mentioned above.

ENABLEMENT

Enablement is indicated in the current enable cells, a two byte field held at offset X'56' in the DSA. Each condition whose enablement is under programmer control has a bit allocated to it. The conditions associated with each bit are shown in figure 7.9.

The CHECK condition has three bits associated with it. This is because the CHECK condition can be used both as a qualified and as an unqualified condition. Bit zero indicates that CHECK is enabled, either qualified for one or more variables, or unqualified for all variables. Bit 11 indicates that CHECK has been enabled or disabled as an unqualified condition. Bit 10, only valid if bit 11 is set, indicates whether the unqualified CHECK is enabled or

Bit	0	CHECK*
Bit	1	ZERODIVIDE
Bit	2	FIXEDOVERFLOW
Bit	3	SIZE
Bit	4	CONVERSION
Bit	5	OVERFLOW
Bit	6	UNDERFLOW
Bit	7	STRINGSIZE
Bit	8	STRINGRANGE
Bit	9	SUBSCRIPTRANGE
Bit	10	CHECK*
Bit	11	CHECK*

* See section "The CHECK Condition" for details

Figure 7.9. Meaning of enablement bits

disabled. (See later section in this chapter "Handling the CHECK condition for further details.)

A further two byte field in the DSA held at offset X'54' is known as the block enable cells. This field is similar to the current enable cells and holds a record of the enablement at the start of the block.

Both current enable and block cells are set up by the prologue code. If the enablement is altered for the duration of a statement, the appropriate bit in the current enable cells is altered at the start of the statement. At the end of the statement the bit is reset to its previous value. If there is an interrupt during the execution of the statement, on-unit action may return control to another part of the block where different conditions are enabled. The block enable cells are necessary to allow for this. Whenever a GOTO out-of-block occurs in an on-unit the GOTO code in the TCA resets the current enable cells from the block enable cells. This ensures enablement will be correct, regardless of the situation when control left the block.

Qualified Conditions

The only qualified condition whose enablement is under programmer control is the CHECK condition. As CHECK is a special case it is treated in detail elsewhere. The principle involved however is that enablement for any particular qualifier is given in a dynamic ONCB and, to discover whether CHECK is enabled for a particular item, a search must be made in the DSA chain for a relevant dynamic ONCB.

ESTABLISHMENT - EXECUTING ON AND REVERT STATEMENTS

For establishment the situation differs between qualified and unqualified conditions. This is because at any one point in the program there can only be one established on-unit for an unqualified condition but there can be an unlimited number of established on-units for qualified conditions. In a program with a number of files, for example, the programmer may wish to take different action when the end of the data is reached in each of the files. Consequently there could be an established ENDFILE on-unit for each file.

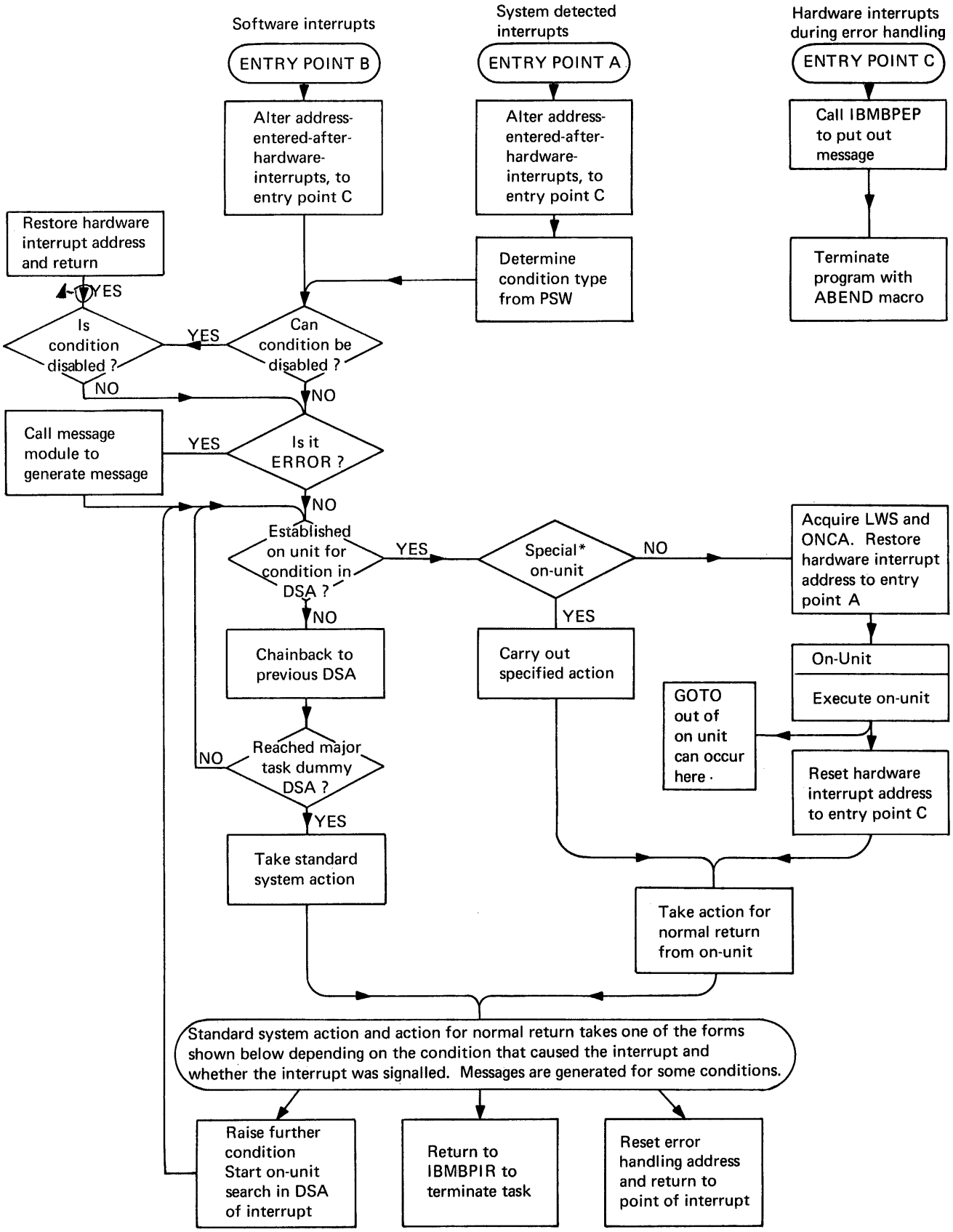
On-units are established by the execution of an ON statement. Once it has been discovered that an on-unit is established it is then necessary to access the on-unit. Access to the address is made through a control block known as the ON-control block ONCB. For unqualified conditions, ONCBs are set up during compilation in static internal storage and are known as static ONCBs. For qualified conditions, ONCBs are set up (by compiled code) in the DSA and are known as dynamic ONCBs. See figure 7.7.

Qualified Conditions

The establishment of qualified conditions is indicated directly in the ONCB. All dynamic ONCBs for a block are chained together and address of first ONCB on the chain is held in a field at offset X'60' in the DSA. (See figure 7.8.)

Dynamic ONCBs contain a code indicating the condition type, flags to indicate whether the condition is enabled and whether the associated on-unit is established, a method of identifying the qualifier, and, either the address of the compiled code on-unit, or flags indicating the action specified in the source program on-unit. There is an ONCB for every ON statement in the block that refers to a qualified condition.

ON and REVERT Statements: When the ON statement is executed the appropriate dynamic ONCB is set up, chained, and the establishment bit in the ONCB is set 'on' by compiled code. For second and subsequent ON statements or REVERT statements for the same condition and qualifier, the information in the ONCB (flags and address of on-unit) is altered.



* Special on-units are not entered these are: null on-units, or on-units containing only a SNAP or SNAP SYSTEM instruction.

Figure 7.10. Simplified flowchart of IBMBERR

Unqualified Conditions

For unqualified conditions establishment information is held in a series of one byte fields known as oncells. There is one cell for each ON statement in the block and, consequently, for each ONCB associated with the block. ONCBs for unqualified conditions are held contiguously in static internal storage in program block order. (See figure 7.8.)

In each DSA containing ON statements an area is reserved for ON cells. Cells are one byte fields that correspond one-for-one with the static ONCBs for that block. The first ONCB for the block is addressed from offset X'5C' in the DSA. ON cells are initialized to zero by the prologue code. When the ON statement associated with the on-unit is executed, a code is set in the ONCELL indicating the condition type. The error handling module searches for an established on unit by testing the ON cells in the DSA of each active block until, either an active ON cell for the condition is found, or the major task dummy DSA is reached. When an active ON cell is found, the number of ON cells in the block preceding the active ON cell are calculated. The associated static ONCB will be in the same relative position. As all ONCBs for unqualified conditions are the same length the address of the requested ONCB can be determined and the action to be taken decided from the ONCB.

ON and REVERT Statement: When an ON statement is executed a code indicating the condition type is set in the appropriate ON cell. If there was a previous ON statement for the condition the former ON cell is set to zero. For REVERT statements any ON cell referring to the condition is set to zero.

If there is more than one ON-statement for the same condition in a block, the flags in the previous ON cell will be set off when second and subsequent ON cell flags are set on. The REVERT statement is executed by setting the flag in the latest ON cell to zero. The situation then reverts to that at the start of the block.

HANDLING ON-UNITS

On-units, except certain single-statement on-units, are treated as separate program blocks by the compiler. They are separated from the ON statement and compiled with prologue and epilogue code. The address of the on-unit is placed in an address constant. The ON statement remains in its logical place in the program and sets either the ON cell or a flag in the dynamic ONCB, to indicate that the associated on-unit is established.

In order to save the overhead of executing prologue and epilogue code, certain single-statement on-units are not compiled. Instead the action required is indicated by flags in the ONCB and is carried out under the control of the error handling module.

The types of on-unit involved are:

1. Null on-units.
2. On-units containing only SNAP, SNAP SYSTEM, OR SYSTEM options.
3. On units containing only a GOTO statement.

The presence of these on-units is indicated by flags in the associated ONCB. For the GOTO only on-unit, the ONCB also contains the offset in the DSA of the label variable or label temporary to which the GOTO is to be made.

The Logic of the Error Handler

A simplified flowchart of the error handling module IBMBERR is given in figure 7.10. This flowchart shows the action during the handling of an interrupt and includes execution of an on-unit. The logic is described below. A complete description is given in the licensed program product document OS PL/I Resident Library Program Logic.

IBMBERR - ERROR-HANDLING MODULE

The error-handling module, IBMBERR, handles three situations. These are:

1. Program check interrupts.
2. PL/I conditions detected by the object program.
3. Errors detected by the object program that are not directly related to PL/I conditions and which raise the ERROR condition.

All three situations are ultimately dealt with as PL/I conditions. For example, the FIXEDOVERFLOW condition would be raised when fixed point overflow occurs and causes a program check interrupt. Where there is no directly-applicable, PL/I condition (for

instance after a data interrupt) a system message is printed and the ERROR condition is raised.

PROGRAM CHECK INTERRUPTS

Before a program check interrupt can be handled as a PL/I condition, action must be taken to prevent the system terminating the job should a further program check interrupts occur. This is done by altering the old program PSW and returning out of the SPIE exit code so that it appears to the system that the interrupt has already been handled. The second word of the PSW passed to ERR in the PIE (program interrupt element) containing the interrupt address is stored in the register 15 field in the save area which was current when the interrupt occurred. IBMBERR then changes the address in the PSW in the PIE to an address in IBMBERR. Control then passes via the supervisor to the address in IBMBERR that has been inserted in the PSW. Handling of the interrupt consequently appears to the supervisor to be finished. The address, in the field in the TCA, to which control will pass after a program check interrupt is then changed to IBMBERRC. Should an interrupt now occur during the execution of IBMBERR, control will pass to IBMBERRC, which terminates the job.

The first task is to generate a suitable error code that will equate the interrupt with a PL/I condition. The floating point registers are saved in IBMBERR's DSA, if the interrupt is one corresponding to a PL/I condition, and control can then be passed to the main PL/I condition-handling routine described in the next section. There are, however, three special cases that require further action. These are:

1. If the interrupt was floating point underflow, then the doubleword in which the floating point register which underflowed was stored is set to zero.

2. If fixed point overflow, exponent overflow, decimal overflow, or fixed point divide has occurred, then it may correspond to the PL/I condition SIZE and not to FIXEDOVERFLOW or ZERODIVIDE. If this is possible, a flag will have been set in the program check interrupt qualifier in the TCA. A test of this flag is therefore made and the necessary action taken, SIZE being raised if it is enabled.

   If the interrupt was an <u>operation interrupt</u> it may have been caused by an extended floating point instruction being used on a machine that does not have the extended float instruction set. If this is the case, the instruction may require simulation. The error handler therefore passes control to a module IBMBEEF that interfaces with the extended float simulator IEXPSIM. IBMBEEF passes control to the extended float simulator which returns the correct result if the statement was valid, or a return code if the statement was invalid. If the statement is valid IBMBEEF returns control to the point of interrupt. If the statement was invalid IBMBEEF returns control to the error handler.

For those installations that do not require extended float simulation a dummy version of IBMBEEF is available. This module returns control directly to the error handler and the error condition is raised.

SOFTWARE INTERRUPTS

When the main condition-handling logic is reached, an error code will have been generated to indicate the type of error or condition that has been raised. For program check interrupts, the code is produced by the error module itself. For errors or conditions detected by the object program, the object program sets up this code. When the object program has detected the error, this will, in some cases, correspond to a PL/I condition. However, there are certain errors (such as attempting to take the square root of a real negative number) that do not have directly-related PL/I conditions. For PL/I conditions, a four-byte code is passed. For other errors, the code consists of only two bytes. For the two byte code, the first byte indicates which class of error has occurred. For the four byte code, the first byte is the identifier of the PL/I condition being raised (the same identifier is used in on-cells).

The error-handling module checks the first byte of the code to see whether it is handling ERROR or another PL/I condition. If the code indicates ERROR, then the message module IBMBESM is loaded into a VDA and called. This module prints the relevant diagnostic message; a suitable four-byte code is then generated. The situation is then treated as for any other PL/I condition.

The second two bytes of the code passed when a PL/I condition has been raised

indicate which condition built-in functions are relevant to the condition. If the condition is one that needs to be qualified, the qualification is also passed.

When a PL/I condition error code is passed, action depends on whether the condition is one of those that can be disabled by the programmer. If it is such a condition, a test is made in the current enable cells of the DSA. If the condition is not disabled, then a search for a relevant established on-unit must be made. If the condition is disabled, a return is made to the point of interrupt. To find established on-units, a test is first made in the action byte to discover whether the condition is qualified. If the condition is not qualified, a search is made through the on-cells of all active blocks to find a match for the number in the first byte of the code passed to IBMBERR. This is done with a translate-and-test instruction using the TRT table addressed from offset X'1C' in the TCA. When found, the position of the located on-cell gives the position of the associated ONCB. A test can then be made to determine the action to be taken.

If the condition is qualified, a search for an active matching ONCB is carried out through the chain of dynamic ONCBs held in the DSAs.

If the major task dummy DSA is reached without a match being found, then standard system action is taken. This action is defined in IBMBERR. When a matching active ONCB is found, tests are then made, as follows, on the flags in the ONCB.

Test 1. SNAP specified? If so, the message module IBMBESM is dynamically loaded and a SNAP message printed.

Test 2. Is SYSTEM specified? (This can occur when "ON condition SYSTEM" has been specified.) If SYSTEM is specified, then the action in IBMBERR is taken.

Test 3. Does the on-unit consist only of a GOTO statement? If so, then the GOTO is executed without entering an on-unit. This saves the housekeeping involved in entering an on-unit.

Test 4. Is the on-unit a null on-unit? If so, then the action on a normal return from the on-unit is taken.

If none of these is positive, then it is necessary to enter the on-unit.

Before entering the on-unit, the following action must be taken. A new allocation of library workspace must be initialized and its address put into the standard offset in the DSA of IBMBERR. This provides workspace for any further library modules that may be called. Tests must be made to see that the ONCA is correctly set-up for any built-in functions that may be used. The address in the PICA field which was altered to the error handler, must also be altered to its original setting so that program check interrupts will cause entry to be made to the error handler by the entry point IBMBERRA rather than IBMBERRC. This ensures that the action specified by the PL/I program is taken if a program check interrupt occurs during the execution of an on-unit.

Normal return from the on-unit to IBMBERR is made by a branch on register 14. Depending on the condition, a return to the interrupted program is then made, or some special action may be taken. Four PL/I conditions cause action other than return to be taken.

1. ERROR
   If the condition was the ERROR condition, then the FINISH condition is raised.

2. FINISH
   If the FINISH condition is raised then a return code is set in the correct field of the TCA, and GOTO performed to the termination routine IBMBPIR. (If FINISH is signalled, then return is made to the point of interrupt.)

3. CONVERSION
   If CONVERSION was raised, then a test is made in the ONCA, and if either ONSOURCE or ONCHAR has been accessed, control is passed to the address contained in the retry slot in the ONCA. The conversion is then attempted again. If the field has not been changed, then the ERROR condition is raised.

4. ENDPAGE
   If ENDPAGE was raised, then a return code is set in register 15 to indicate that an on-unit has been entered.

RETURN TO POINT OF INTERRUPT

Software Interrupts

If the condition was one that was detected by compiled code, then a return to the point of interrupt is made by a branch on

register 14.

Program Check Interrupts

For program check interrupts, the status of the program at the original point of interrupt has to be restored before return to the point of interrupt can be made. This means that the contents of the system save area must be reset, so that they are identical with those saved after the original interrupt. (The PSW and the register values were saved in the DSA at initial entry to IBMBERR.)

The method used is as follows. The address in the PICA is altered so that the address that is to be branched to, after a program check interrupt, is changed from IBMBERRC to another point in IBMBERR. An interrupt is then caused, and the supervisor gains control. Consequently, the address in IBMBERR is reached with the address of the system save area in register 1. The contents of the save area and the PSW are then changed to those that were current after the original interrupt. The point of entry for program check interrupts is then reset to IBMBERRA. Return is made to the address in the PSW, which is that of the original interrupt.

THE CHECK CONDITION

The CHECK condition has to be handled in a different manner to other conditions. This is because it can be used as a qualified or unqualified condition and its enablement is under programmer control.

The CHECK condition is disabled by default and is enabled by writing a CHECK prefix. It can be disabled for the duration of a statement or block by the NOCHECK prefix. Prefixes can take the form (CHECK) or (NOCHECK), or the form (CHECK(A,B)) or (NOCHECK(A,B)). When no name list is appended, the CHECK applies to all the relevant names in the program. An ON-statement may also be written as either ON CHECK or ON CHECK(A,B). ON-statements are independent of prefixes and may be included in a block to which no prefix applies. A qualified on-unit can be used with an unqualified prefix and vice-versa.

Throughout this discussion, CHECK and NOCHECK without a name list are referred to as unqualified. CHECK or NOCHECK with a name list are referred to as qualified.

Raising the Check Condition

CHECK is normally raised by compiled code. This is done by inspecting the source program and generating calls to the error handler at appropriate points. As enablement is statically descendent, it is possible to tell during compilation at which points CHECK is enabled and consequently at which points the calls to the error handler have to be made. However, for GET DATA statements there is no means of knowing which items will be passed in the data stream, and if the CHECK condition is enabled for any variable that could be read in, it is necessary to check every variable in the input stream to see whether CHECK is enabled for that variable. Consequently, when a GET DATA instruction is being executed, it is necessary for the error handler to test to see if the CHECK condition is enabled.

With the exception of the CHECK condition, all conditions whose enablement is under programmer control are unqualified. Consequently, their enablement or disablement can be indicated by one bit in the enable cells. This is because there are only two possibilities. Either the condition is enabled or it is disabled. With qualified CHECK, however there are many possibilities, because CHECK may be enabled for some variables and disabled for others. Consequently, the enable cells are used in a different manner for the qualified CHECK condition, and the enablement of qualified CHECK for any particular name is given in an ONCB.

When the CHECK condition is raised, the error handler has the following tasks.

1. Test to see if CHECK occurred during the execution of a GET DATA statement. If so tests for enablement must be made. If not continue with step 3.

2. Test to see if CHECK is enabled. This involves a search along the static backchain to determine, for each block, first, if qualified CHECK is enabled or disabled for the particular name for which CHECK was raised, and then, if unqualified CHECK is enabled or disabled.

3. Search for a qualified established on-unit. This involves searching the dynamic backchain for a relevant dynamic ONCB.

4. If there is no qualified established on-unit search for an unqualified established on-unit. This involves a further search of the dynamic backchain looking for appropriate

on-cells.

5. If no established on-unit is found, take standard system action.

This process is illustrated in figure 7.11.

Testing for Enablement

There are three bits that refer to CHECK in the enable cells; they have the following significance:

Bit 0

'0'B CHECK is enabled for certain items in this statement

'1'B CHECK is disabled for this statement

Bit 10 (only valid if bit 11 is set)

'0'B The unqualified prefix that applies is NOCHECK

'1'B The unqualified prefix that applies if CHECK

Bit 11

'0'B No unqualified prefix applies to this statement

'1'B An unqualified prefix applies to this statement

Throughout this discussion Bit 0 is referred to as the "any-CHECK" enablement bit, and bits 10 and 11 as the "unqualified CHECK enablement bits." Enablement and disablement of qualified CHECK is indicated in the flag bits of the ONCB.

The test for enablement begins by a test on the any-CHECK bit in the enable cell. If this is set to zero, control is immediately returned to the caller. If the bit is set on, a search is made for a relevant qualified ONCB in the DSA of the block in which the interrupt occurred. If no such ONCB is found, the unqualified CHECK enablement bits are tested for unqualified enablement or disablement. If bit 11 is not set, neither an unqualified CHECK nor an unqualified NOCHECK applies, and a further search must be made in the preceding DSA on the static backchain. If the dummy DSA is reached without any of the tests proving positive, CHECK is disabled.

Searching for Established On-Units

When it is known that CHECK is enabled, a search must be made for established on-units. This search is separate from the search for enablement. A return is first made to the DSA in which the interrupt occurred.

Two searches are made, the first for a qualified on-unit. The complete dynamic backchain is searched for relevant ONCBs. If one is not found, a search is made through the backchain for enable cells that indicate unqualified CHECK. If nothing is found, standard system action is taken.

Standard System Action

Standard system action for CHECK is taken under the control of a special module IBMBERC. This module acquires the necessary symbol table address or addresses, places them in a VDA and passes control to the stream I/O initializing routine and, on return, to the data directed director module IBMBSDO. On completion of the operation IBMBERC returns control to IBMBERR.

Error Messages

The library module IBMBESM is called by the error handler to transmit the system messages and find the on-code value by calling the ONCODE routine IBMBEOC; control is then passed to IBMBESN to finish the system message, or to go to generate the SNAP message if required. The text for the messages is taken from a series of message text modules. The particular message text module required and the message within the module are determined from the error code.

Message Formats

System Messages: For non-PL/I conditions, system messages have the following form:

```
IBMxxxx 'ONCODE'= xxxx message text
[qualifier] IN STATEMENT xx AT/NEAR
OFFSET xxx IN PROCEDURE WITH ENTRY
xxxx
```

The qualifier might, for example, consist of the file name. For PL/I conditions, the format of the message is much the same, but

the name of the condition is also given. For example:

```
IBM4021 'ONCODE'= 3100 'FIXEDOVERFLOW'
CONDITION RAISED IN DECIMAL DIVIDE IN
STATEMENT 31 AT OFFSET 000A35 IN
PROCEDURES WITH ENTRY ZERNES
```

Snap Messages: If an on-unit contains both SNAP and SYSTEM, the resulting message is essentially the system message followed by the line

```
FROM (STATEMENT/OFFSET) xxx IN A
(BEGIN BLOCK/PROCEDURE WITH ENTRY
xxx/A 'xxxx' ON-UNIT)
```

which is repeated as many times as necessary to trace back to the main procedure. If an on-unit contains only SNAP, the message begins

```
'xxxxxxx' CONDITION RAISED [IN
STATEMENTxxx] (AT/NEAR) OFFSET xxx IN
PROCEDURE xxx
```

and continues as for a SNAP SYSTEM message.

The statement number is not always present in messages as the generation of execution-time statement numbers by the compiler is a compiler option.

When statement numbers are generated, they are held on a block basis. For each block or procedure, a table in static storage relates each statement number to the offsets of the corresponding instructions in compiled code. A field at a fixed offset each entry point gives the address of the relevant table.

The statement number is held in relation to its offset from the main entry point. Since the PL/I program need not have entered via this entry point, the offset is calculated independently from that given in the message. If the FLOW option is used, then additional information is printed out after every snap message. (See "The FLOW Option," later in this chapter.)

Interrupts in Library Modules

When an interrupt occurs in a library module, the system message does not give the offset from the start of the library module, but gives the statement number of the statement in which the library module was called and the offset of this statement from the entry point of the procedure block in which it is contained.

Identifying the Erroneous Statement

The address required to identify the erroneous statement is always the address held in the register 14 field in the most recent compiled code DSA.

If the interrupt was a software interrupt in compiled code, the address will be the return address that was used by the BALR instruction when IBMBERR was called.

If the interrupt was a program check interrupt in compiled code, the address of the interrupt will have been moved from the old PSW and placed in the register 14 field by IBMBERR to simplify return to the point of interrupt.

If the interrupt was in a library module, the address required is the point in compiled code at which the library routine was entered. This will have been placed in the register 14 field when the library module was called.

Identifying Entry Point Name and Statement Number

The address of the entry point of the block is found by chaining back along the DSAs to the DSA before the last compiled code DSA. The address of the entry point used before the interrupt is held in the save area of this DSA as the branch register contents. The dummy DSA ensures that a chainback can be made from the main procedure DSA.

The name of the entry point is found by chaining back one DSA beyond the first procedure-DSA reached. This DSA holds the address of the procedure-DSA entry point in the register 14 slot of its register save area (offset X'10' from the head of the DSA). The length of the name is held in a one-byte field immediately preceding the entry point. The name immediately precedes the length field.

Statement numbers are generated separately for each external procedure, and the statement number table holds offsets from the first entry point in the external procedure.

When the statement number table is link-edited, the address of this entry point is placed at the head of the table. Consequently, the required offset can be found by comparing the address of the statement causing the error with the address of the first entry point held in the statement number table.

If the NUMBER option is in force, the numbers are held in four byte form preceded by a halfword statement number. Otherwise, the statement numbers are held in two byte form. Flags indicating which options are in use are held in the DSA. They are shown in appendix A.

As the offsets may be up to 6 bytes in length, a device is used for statement numbering whereby the table is divided into sections that correspond to the offset values that are held in the first two bytes of the offsets. Thus offsets starting X'00' are held in the first section of the table, offsets starting X'01' in the second, and so on. Each section of the table is headed by a pointer to the start of the following section, or set to zero if there is no following section. The complete table is also headed by the value of the maximum offset, so that offsets beyond the program can be readily detected.

The statement number is found by searching the correct section of the table for the first offset that is less than or equal to the last four hexadecimal digits of the calculated offset.

For snap messages, once the on-unit has been found and the appropriate message generated, the rest of the trace gives information about procedures, begin blocks and on-units. Thus all compiled code DSAs can be treated in the same way.

Filename and Name of CONDITION Condition

If the error was in I/O, then the address of the DCLCB of the file is passed to IBMBERR which stores it for IBMBESN to find the file name. Similarly, the address of the control section containing the condition name is passed to IBMBERR if the CONDITION condition is raised, and IBMBESN puts out the required section of message.

MESSAGE TEXT MODULES

The message module IBMBESM calls on a number of message text modules to produce the relevant message. These modules consist essentially of the fixed message text portions of the message. The messages are held in groups.

The groups are addressed from a table at the head of the module, and the messages in their turn are addressed by an offset from the start of each particular table in the message text modules. The message required is determined from information in the error code. IBMBESN puts all error messages onto SYSPRINT provided that SYSPRINT has nct been declared with unsuitable attributes. If it has been declared with unsuitable attributes, then the system messages go to the console operator, and the snap messages are ignored.

DIAGNOSTIC FILE BLOCK

Every attempt is made to put out error messages on the standard print file SYSPRINT. However, there are no reserved words in PL/I and consequently the name "SYSPRINT" may be used for a file with attributes other than PRINT OUTPUT, or may be used for a variable of any other data type. If SYSPRINT is declared as an unsuitable type of file it cannot be used for error messages and all error messages are written on the console.

A control block, the diagnostic file block (DFB), is set up during program initialization to indicate whether SYSPRINT can be used for error messages. If SYSPRINT has been declared as a file the address of the DCLCB is placed in the DFB. The DFB (diagnostic file block) is addressed from the TCA. When an error message module is to be put out IBMBESM or IBMBPEQ inspects the DFB to see if SYSPRINT can be used for the message. If the flags in the DFB indicate that SYSPRINT cannot be used, the module IBMBEDO is called.

IBMBEDO tests to see if SYSPRINT is open if it is not, calls IBMBOCL to open it with the attributes STREAM PRINT. If SYSPRINT has been declared as a file the address of the DCLCB is picked up from the DFB. Should the attributes STREAM and PRINT be incompatible with the declared or default attributes this is diagnosed by the OPEN module and Appropriate flags are set in the DFB to indicate that SYSPRINT cannot be used for error messages. This action does NOT raise the error condition.

If SYSPRINT has not been declared, a DCLCB will be generated and SYSPRINT will be opened, provided that the error occurs before a task has been attached. If a task has already been attached, or if the error occurs in an attached task, then SYSPRINT cannot be opened and all error messages are passed to the console.

If SYSPRINT is already open with unsuitable attributes this will have been flagged in the DFB and the messages will again be passed to the console.

If SYSPRINT has been declared as a data type other than a file this is flagged in the DFB and the error messages are set to the console.

If SYSPRINT has not been declared at all, a diagnostic SYSPRINT is opened and used, provided that there is a DD card for SYSPRINT.

Dump Routines

A series of library modules are provided to implement the PLIDUMP facility. Module IBMBKDM is the dump bootstrap module which is part of the resident library. This loads and calls the transient dump control module IBMBKMR, which in turn links and calls those modules required to carry out the dump options specified in the call to PLIDUMP. Several transient modules are used to reduce the amount of storage used at any one time. The organization of these modules is shown in figure 7.12.

In order to ensure that as much information as possible is provided when a call to PLIDUMP is made, a special SPIE macro instruction is issued at the start of every transient routine to intercept program check interrupts during the routine. When a program check interrupt occurs, an attempt is made to continue with the dump. If the interrupt occurs in a program called from the dump control module, that particular routine is abandoned and a return is made to the dump control module. Any further routines needed to complete the information specified in the options are then called. If the interrupt occurs in the trace or file modules, the "H" option is assumed and a hexadecimal dump produced. If the interrupt occurs during the execution of the hexadecimal dump module, a SNAP macro instruction is issued by the dump control module and a snap dump is completed under the control of the supervisor. When the snap dump is completed control returns to the dump control module and the PLIDUMP is completed as requested in the dump options.

As further insurance against error, the dump control module IBMBKMR is divided into sections, and, if an interrupt occurs in any of these sections, control is passed to a predefined address at the end of the section. Processing then continues from that point.

The dump modules are fully described in the publication OS PL/I Transient Library Program Logic.

Dump File

In order to avoid mixing of PL/I dump and other information, dump data is not transmitted to any PL/I file. A special dump file known as PLIDUMP is used for the output of the dump modules. This file has its own transmitter and a special opening module IBMBKDO. A control block, the dump block, (DUB) is set up during program initialization and is used to hold information about the status of the dump file and to simplify access to the file. The DUB (dump block) is addressed from offset X'24' in the TCA appendage. To generate a PL/I dump it is necessary to have a DD card for PLIDUMP, or PL1DUMP.

Before any output has been produced by the dump modules, the dump control module IBMBKMR inspects the DUB to see if the dump file is open. If the dump file is not open, and is not flagged as unopenable, the control module calls the dump file open routine (IBMBKDO) to open the file. IBMBKDO acquires space for the necessary control blocks loads the dump transmitter and attempts to open the dump file.

If the attempt to open the dump file fails, IBMBKDO flags the DUB and returns. The DUB flags are tested by IBMBKMR, and, if the file has not opened, a message is put out and the dump is terminated. The job is either continued, terminated or an exit is made from the task, according to the options in the dump parameter. IBMBKDO uses either the declared PLITABS or loads the system default PLITABS module, IBMBSTAB to determine the pagesize for PLIDUMP output. Provided a pagesize of two or more is specified, the pagesize in PLITABS will be used.

If the dump file can be successfully opened, IBMBKDO tests the attributes of the file. It it appears from the attributes that the dump is being transmitted directly to a printer or terminal,, the transmitter IBMBKDT is loaded. If it appears that it is being transmitted to a direct-access device or tape unit, the transmitter IBMBKDB is loaded.

If IBMBKDT is loaded, two buffers are acquired. The address of one of these buffers is placed in the DUB. During the execution of the dump, the dump data is generated in the buffer which is addressed by the DUB. When the first buffer is full, a call is made to the transmitter module to transmit the buffer to the dump file. A test is then made to see whether the second buffer has completed the previous I/O operation. When the previous I/O operation

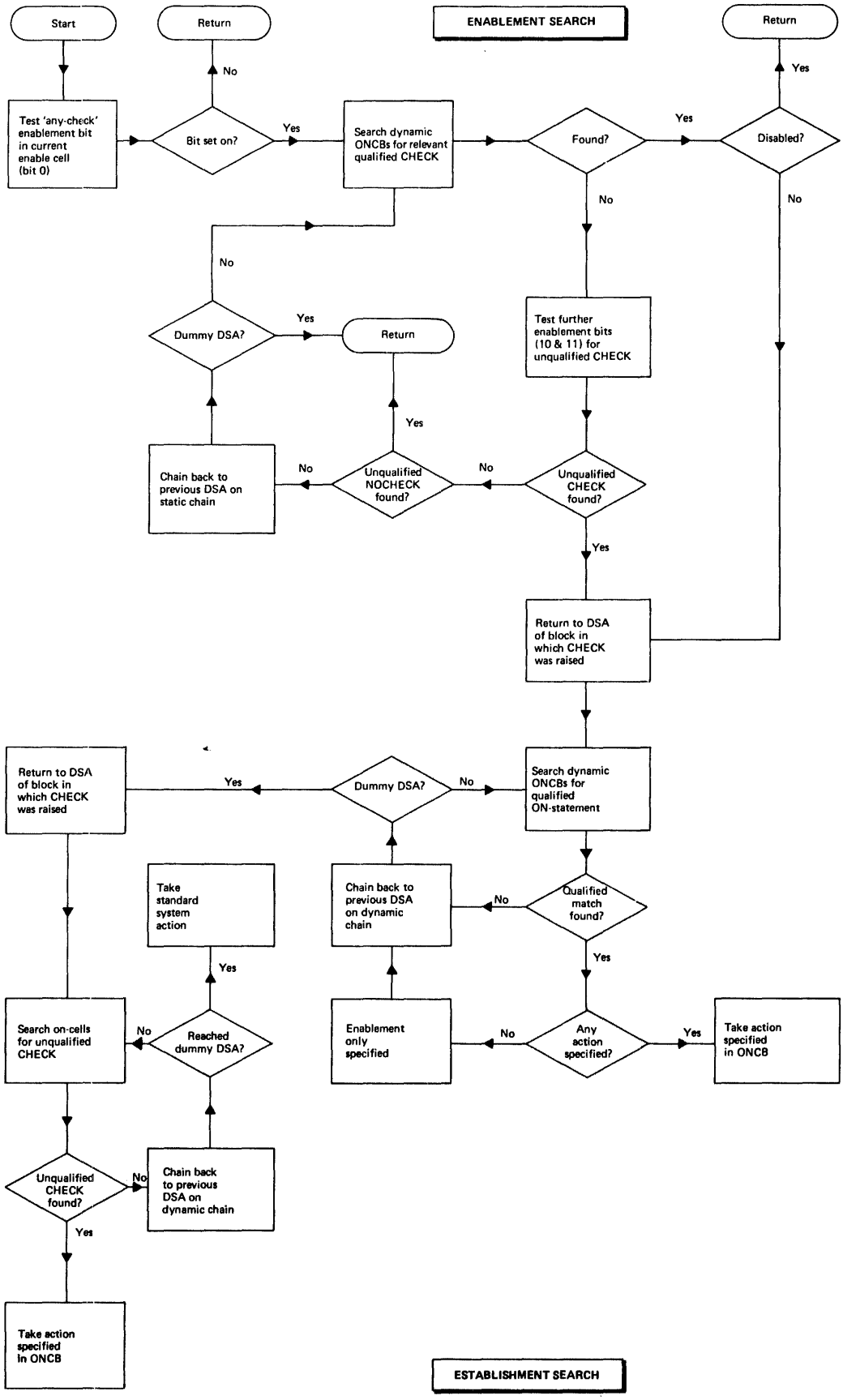


Figure 7.11. Handling the CHECK condition

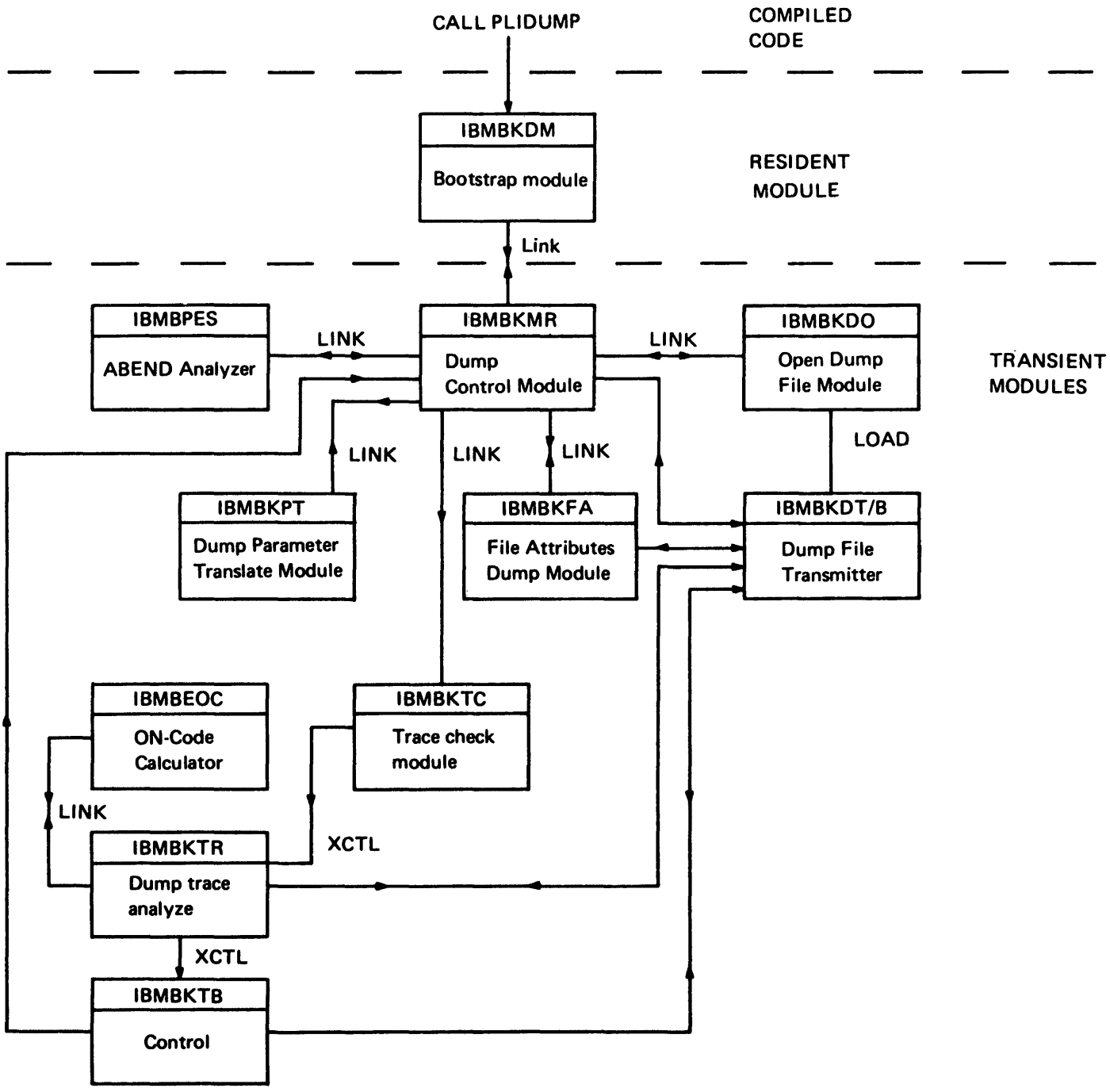


Figure 7.12. Interrelationship of dump routines

(if any) is complete the address of the second buffer is placed in the DUB and the operation continues. If IBMBKDB is loaded, only one buffer is used.

When the dump is finished, the dump file remains open and the transmitter is retained. This speeds execution of further dumps. The storage is freed and the dump file closed by IBMBPIT when the program is terminated. The dump file is not placed on the open file chain. IBMBPIT tests the DUB to see if the file is open.

Miscellaneous Error Modules

A number of further library modules are used in certain exceptional error situations. These fall into two groups.

1. ABEND analyzers

   IBMBPES Determine action to be taken.

   IBMBPEV Put out message if necessary, and dump if possible.

2. Exceptional error message modules

   IBMBPEP Exceptional error message director

   IBMBPEQ No main procedure or more than 1024 files and controlled variables.

   IBMBPER No main storage available

   IBMBPET Interrupt in error handling routines or abnormal task termination

All these modules are transient library modules. They are fully described in the relevant program logic manual.

Abend Analyzers

The ABEND analyzer IBMBPES is entered during an ABEND because it was nominated in the STAE macro instruction issued during program initialization.

The ABEND is analyzed by checking the major blocks to see if they have been overwritten. If the backchain of DSAs has become overwritten, the ABEND is allowed to continue under supervisor control. If the DSA backchain is correct but critical control blocks appear to be overwritten, IBMBPEV is called to put out a message and if possible to provide a PLIDUMP. If no overwriting is detected, the error handler is called with a code indicating the error condition.

The message put out by IBMBPEV where possible contains the number of the PL/I statement being executed when an ABEND occurred.

EXCEPTIONAL ERROR MESSAGE MODULES

The exceptional error message modules consists of a director and three message modules. This arrangement has been adopted so that the minimum space will be used. It is necessary to conserve space as lack of space is one of the reasons for calling the modules.

The director module IBMBPEP determines the nature of the exceptional error and calls the necessary module to put out the message.

The table below shows the circumstances in which IBMBPEP is called and message modules then called by IBMBPEP.

Circumstance	Calling module	IBMBPEP calls
Insufficient main storage to set up program management area	IBMBPII	IBMBPER
No main procedure	Code in dummy PLIMAIN	IBMBPEQ
Too many files or controlled variables to be held in PRV	IBMBPII	IBMBPEQ
Interrupt in error handling routine	IBMBERR entry point C	IBMBPET
Abnormal termination of task	IBMTPIR	IBMBPET

Module IBMBPEQ puts out the the message to SYSPRINT except in those circumstances where SYSPRINT cannot be used. (See above under "Error Message Modules"). IBMBPET and IBMBPER always put out their messages on the console as they are called in circumstances where SYSPRINT is likely to fail or where operator, rather than programmer action, is required.

PL/I PROCEDURE TO BE COUNTED

```
  1 COUNTIT:PROC OPTIONS (MAIN);
  2           DO I=1 to 2;
  3                PUT LIST (I);
  4           END;
  5         END COUNTIT;
```

In this procedure, the do-loop in statements 2 through 4 will be executed twice, and the other statements once. Statement 2 will be executed three times as a return is made at the end of the loop to test the value of I. Note: This code may compile in different ways. See section on DO-loops in chapter 2.

HISTORY OF THE STATEMENT FREQUENCY COUNT TABLE

After the branch-in to statement number 1, the table is set up with a value of 1 for the first statement and 0 for all others, thus:

statement number	1	2	3	4	5
branch count	1	0	0	0	0

After the branch-out at statement 4, the count of the next statement is decremented by one and the table becomes:

statement number	1	2	3	4	5
branch count	1	0	0	0	-1

After the branch-in at statement 2, the branch count for statement 2 is incremented by one and the table becomes:

statement number	1	2	3	4	5
branch count	1	1	0	0	-1

At statement 4, a further branch out is made and a return made to statement 2 to test the value of I. One is subtracted from the value of statement five making the count -2 and one added to the count of statement 2 making it 3. Because I is greater than 2 a branch is made after the test to statement 5. This results in one being subtracted from the count for statement 3 and one being added to the count for statement 5. At the end of the program the table reads:

statement number	1	2	3	4	5
branch count	1	2	-1	0	-1

ANALYSIS OF THE STATEMENT FREQUENCY COUNT TABLE

A value known as the current count, which is initially set to zero, is added to the branch count for each statement in turn. The sum is the number of times the statement was executed; this value also becomes the current count.

statement number	current count	branch count	times executed
1	0	1	0+1= 1
2	1	1	2+1= 3
3	2	0	3-1= 2
4	2	0	2+0= 2
5	2	-1	2-1= 1

Figure 7.13 How branch counts are used to calculate the number of times each statement is executed.

The FLOW and COUNT Options

The FLOW and COUNT options are used to provide information about which statements are executed in a particular run of a program. The FLOW option is used to maintain a trace of the most recently executed statements. The COUNT option is used to maintain a count of the number of times each statement is executed.

Both options are implemented by calling an interpretive library routine, IBMBEFL, at every point in a program where the flow of control may not be sequential. The library routine, IBMBEFL, analyzes the situation and updates tables to retain a record of the branches made. IBMBEFL is also called during program initialization to set up housekeeping information. Two transient library modules are used to interpret the tables set up by IBMBEFL and to put out the information. The routines are IBMBESN for the FLOW option, and IBMBEFC for the COUNT option.

The compiler generates the same executable code for both the COUNT and the FLOW option. Consequently, if either option is specified for compilation, either or both can be made available at execution time. If neither is required during execution but one or other was specified for compilation, the code to call IBMBEFL is still executed and IBMBEFL still forms part of the load module. When IBMBEFL is called in this situation, it returns control to compiled code without recording any information.

Points at which the flow of control may not be sequential are known as branch-in and branch-out points. For example, labeled statements and entry points are branch-in points, and GOTO statements are branch-out points. At branch-in and branch-out points the compiler places code that will call IBMBEFL. If the branches are taken, they are recorded. For COUNT they are recorded in a table known as the statement frequency count table. For FLOW, they are recorded in a table known as the flow statement table.

Use of Branching Information for FLOW

For the FLOW option, a list of the statement numbers at which branches were taken and a list of any changes of procedure is retained.

FLOW output consists simply of the list that is recorded by IBMBEFL and typically takes the form shown below.

```
        12 TO 18
        27 TO 35 IN SORTER
        76 TO 108 IN TESTER
        134 TO 77 IN SORTER
```

This indicates that the program branched from statement 12 to statement 18, then ran sequentially from 18 to 27. After statement 27 it branched to, or called, statement 35 in the procedure called SORTER. Control then ran sequentially to statement number 76, at which point it passed to statement number 108 in the procedure called TESTER. Control then ran sequentially from 108 to 134 and finally passed to statement 77 in SORTER.

Use of Branching Information for COUNT The COUNT option calculates the number of times each statement is executed by recording branch-in and branch-out points as they occur and analyzing them at the end of the program.

The formula used for calculating the number of times each statement is executed from the branch count is:

Cn=Cn-1+BIn-BOn-1

Where:

- Cn =the number of times the statement was executed.
- Cn-1 =the number of times the previous statement was executed.
- BIn =the number of times the statement was branched to.
- BOn-1=the number of times the previous statement was branched from.

To retain the information, a count field is set up for every statement in the program, and branches-in and branches-out are recorded when they occur. Every time a branch-in is made, the count for the statement to which the branch is made is incremented by one. Every time a branch-out is made, the count for the statement after the branch-out is decremented by one. When the program ends, statements that have values other than zero mark the beginning and end of ranges of statements that have been executed the same number of times. The number of times the ranges of statements have been executed is calculated by adding the value in the count field to the sum of any preceding values.

This process can be followed in figure 7.13.

Special cases There are a number of special cases that require additional action, either by the compiler, or by IBMBEFL, or by both. These special cases arise for three reasons:

1. Branches can be caused by interrupts, but the points at which they will occur cannot be

predicted during compilation. Consequently the compiler cannot place calls to IBMBEFL at these points.

2. Branches to labeled statements, can come from either the same block or a different block. Consequently the code generated by the compiler cannot be used to indicate whether a new block entry is required.

3. The algorithm used for the COUNT option is not effective for CALL statements and function references because the branch-in and branch-out are made to and from the same statement.

The first case is handled by IBMBEFL checking for the occurrence of an interrupt when it is called in situations where one could have occurred. The second case is handled by altering the GOTO code in the TCA so that it calls IBMBEFL to set appropriate flags when a GOTO out of block occurs. A test for the flags is made when the call to IBMBEFL for the branch-in at the labeled statement is made. The third case is predictable during compilation and is handled by the compiler setting up different code for branches-in to CALL statements and function references, and by IBMBEFL testing for such code. Details of the methods used are given later.

IMPLEMENTATION OF FLOW AND COUNT

Tables Used by FLOW and COUNT

To enable it to retain FLOW and COUNT information, IBMBEFL sets up tables in dynamic storage. Figure 7.14 shows their contents. Details of their formats are shown in appendix A. -43

FLOW Option: FLOW information is retained in a table called the flow statement table. The flow statement table has three sections; a header section containing housekeeping information, a statement number section holding the numbers of statements that were branched to or from plus flags to indicate the type of entry, and a procedure names section containing the names of procedures and on-units to which branches are made. The length of the flow statement table is determined by the values given to "n" and "m" when the FLOW option is specified.

When all the spaces in the table for statement numbers or procedure names have been filled, the earliest entries are overwritten. The fields in the header section are used to indicate which is the next space available in the table.

The table is set up during program initialization and is addressed from the TCA.

COUNT Option: COUNT information is retained in tables called statement frequency count tables. The tables have a field for every statement. They are set up when an external procedure is entered. A table is needed for every external procedure because two external procedures can contain the same statement numbers.

Statement frequency count tables are chained together and addressed from the TCA appendage (the TIA). Two addresses are kept in the TIA, the address of the current statement frequency count table (that is the table that was last used) and the address of the statement frequency count table for the first procedure in the chain. Statement frequency count tables are associated with their matching external procedures by having the address of the static control section for the procedure placed at a fixed offset in the table. (A static control section is unique to an external procedure and its address can be easily accessed as it is addressed throughout compiled code by register three). The last statement frequency count table in the chain has its chaining field set to zero.

The length of statement frequency count tables depends on whether the GOSTMT or GONUMBER option is in effect. For GOSTMT one fullword is used for each statement in the procedure. For GONUMBER, two fullwords are used. This is because for GONUMBER it is necessary to retain the statement number as well as the count value. (For GOSTMT, the numbers will start at one and be incremented by one, and no record need therefore be kept.) If neither GOSTMT nor GONUMBER is in effect, no attempt is made to count the statements executed in the procedure and a statement frequency count table is not set up.

Executable Code for FLOW and COUNT

As described in the introduction, there are four stages in the implementation of the FLOW and COUNT options. These are:

1. Action during compilation. The code to call the interpretive library routine IBMBEFL

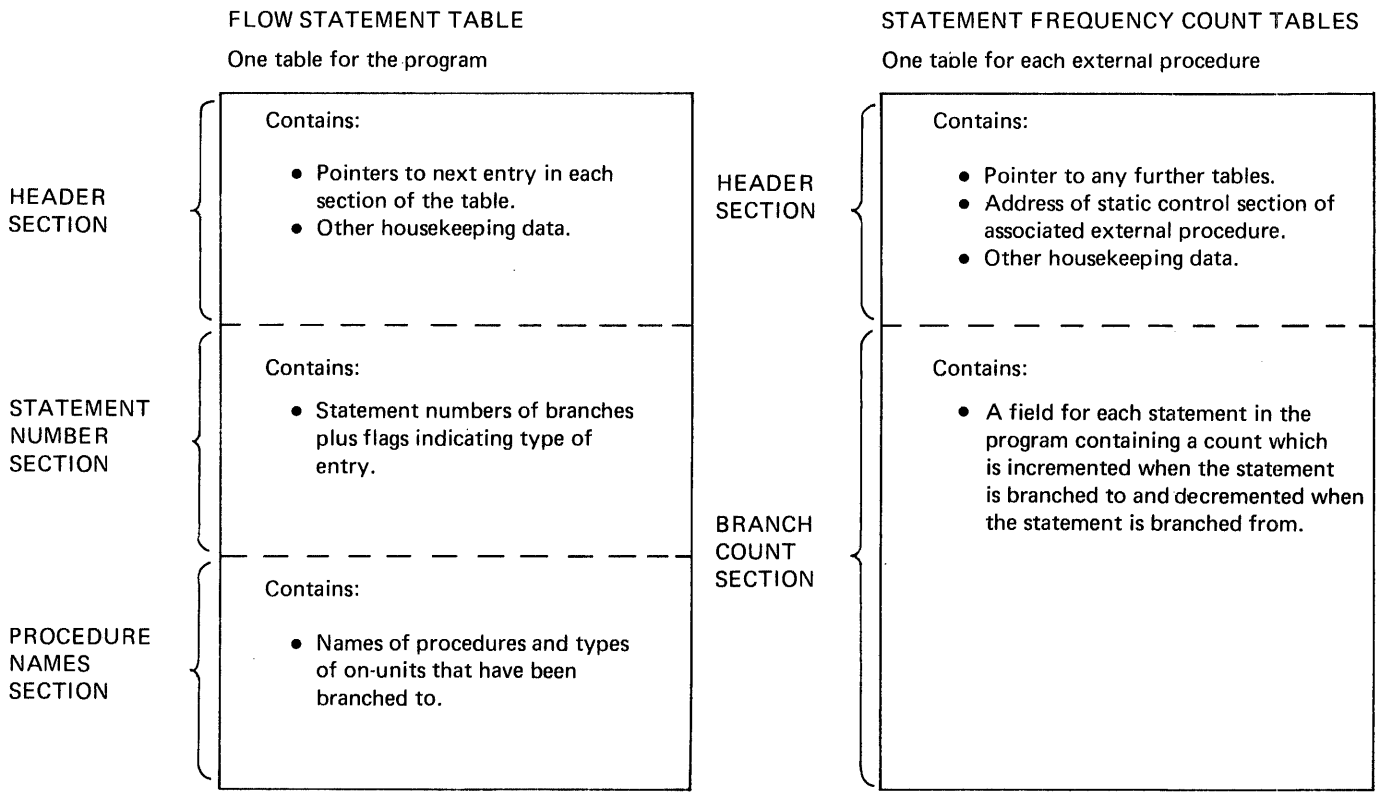


Figure 7.14. The contents of the flow statement table and the statement frequency count table.

is placed at every predictable branch-in and branch-out point.

2. Action during program initialization. The necessary housekeeping fields are set up. This is done by the program initialization module IBMBPII and the flow module IBMBEFL called at entry point A.

3. Action during execution. The branch-in and branch-out information is collected by IBMBEFL, called at entry point B. IBMBEFL is also called at entry point C to handle certain special cases. The call is made when the GOTO out-of-block code is executed.

4. Action during output. The necessary information is written out. This is done by IBMBESN for the FLOW option and IBMBEFCA for the COUNT option.

These four stages are described in detail in the following sections.

Action During Compilation

During compilation, the compiler examines the program and generates suitable code at each predictable branch-in and branch-out point. Predictable branch-in points are:

- entry names
- labeled statements
- THEN and ELSE clauses of IF statements.
- entries to on-units
- returns from CALL statements or function references.
- the statement following the END statement of an internal procedure.

Predictable branch-out points are:

- GOTO statements
- function references
- CALL statements
- IF statements
- RETURN statements
- END statements
- the statement before the PROCEDURE statement of an internal procedure.

The code for branch-out points is so placed that the call to IBMBEFL will not be made unless the branch is taken.

Statements preceding and following internal procedures are treated as branch-out and branch-in points because the statement numbers of the statements executed are not sequential although the actual flow of control is sequential. If this were not done, the method used for counting statements would not work because the statements in the internal procedure would be given the count values of the preceding statements.

The code placed at the branch-in and branch-out points takes the following form:

```
L     15, 84(0,12)        Pick up address of
                          IBMBEFL from TCA.

BALR      14, 15          Branch to IBMBEFL.

DC    X'8004'             Constant containing
                          a two-bit flag
                          remainder for
                          statement number.
```

Register 14 is set to the constant containing the statement number and flags by the BALR instruction. IBMBEFL can therefore pick up the statement number by examining the constant.

The constant is a halfword if the STMT option was used and a fullword if the NUMBER option was used. In both cases, the first two bits are used as flags and the remainder is used for the statement number.

The flags indicate:

- branch-in
- branch-out
- branch-in to a new procedure or on-unit.
- return to point of interrupt from end of on-unit.

For a branch-in to a CALL statement or a function reference, which takes place when the return is made, BAL 14, 0(15) is generated instead of BALR 14, 15. This situation requires to be recognized because the branch-in and branch-out both occur from the same statement. If it were not treated as a special case, the count of the next statement would be decremented by one when the branch-out was made and the count for the CALL statement would be incremented by one on return. Thus the CALL statement would apparently have been executed twice. The increment is therefore added to the statement after the CALL statement, thereby giving the correct values.

In addition to the calls to IBMBEFL, the compiler also generates control sections that will result in IBMBEFL being link-edited and subsequently called during program initialization to set up the necessary housekeeping machinery to handle

COUNT or FLOW.

For the FLOW option, the compiler generates a control section called PLIFLOW that can be used during program initialization to call IBMBEFL. This control section takes the following form:

```
         USING *,15
         L     15,VCON
         BALR  1,15
         DC    H'n'
         DC    H'm'
VCON     DC    V(IBMBEFLA)
```

For the COUNT option, the compiler generates a control section called PLICOUNT that can be used to call IBMBEFL to initialize the COUNT option. It is the same as PLIFLOW except that the halfwords 'n' and 'm' are replaced by a fullword X'80000000'.

The calls to IBMBEFL are generated if either FLOW or COUNT is defined at compile-time. The control sections are generated if the corresponding option is specified at compile time.

Action During Program Initialization

During program initialization, the program initialization module IBMBPII determines if either FLOW or COUNT or both are required. If the user specified either FLOW or COUNT during compilation, the requested option will be in effect during execution unless specifically overridden by the NOFLOW or NOCOUNT execution time option. If he specified either option for compilation he can also specify the other for execution.

To determine which options are to be used, IBMBPII inspects the execution time options and checks for the presence of PLIFLOW or PLICOUNT which will indicate that the corresponding option was requested at compile-time.

If one or both of the options are requested for execution but neither was requested for compilation, IBMBPII generates a message to say that the option will not be available.

If an option is specified for compilation and not overridden for execution time, the corresponding control section will be available and IBMBPII passes control to IBMBEFL at entry point A through the code in the control section. If the control section corresponding to the required option does not exist, IBMBPII calls IBMBEFLA directly, passing it a value in register 0. This value is 4 if FLOW is required and 8 if COUNT is required.

If one or both of the options have been requested during compilation but neither are required during execution, IBMBPII sets FLOW values of (0,0) and calls IBMBEFLB to initialize the FLOW option. In this situation, IBMBEFL sets the address of the flow statement table and the addresses of the statement frequency count tables to zero.

To initialize FLOW, IBMBEFLA sets up the flow statement table and initializes it with a dummy statement number entry and a dummy procedure name entry. The address of the flow statement table is placed in the TCA. If FLOW is not required, or if FLOW(0,0) has been specified, the address is set to zero.

To initialize COUNT, two addresses in the TIA are initialized. The first, which will contain the address of the first of the chain of statement frequency count tables, is set to zero. The second which will contain the address of the current statement frequency count table is set to point to the first. If COUNT is not required, both fields are set to zero.

For both FLOW and COUNT, the address of entry point B of IBMBEFL is placed in the TCA; the GOTO code, which is in the TCA, is altered so that it calls IBMBEFL at entry point C. (This is necessary so that changes of block caused by GOTO statements can be intercepted and flagged.)

Action During Execution

During execution, calls from compiled code at branch-in and branch-out points are made to entry point B of IBMBEFL whose address has been placed in the TCA. The action then taken depends on which options are in effect, the type of the previous entry, and the type of the present entry.

Calls are also made to IBMBEFL at entry point C when the GOTO code in the TCA is executed.

IBMBEFL When Called at Branch-In and Branch-Out Points

When IBMBEFL is called at branch-in or branch-out points, the call goes to entry point B whose address has been placed in the TCA during program initialization. IBMBEFL first checks to see which, if either, of the options is required by testing the fields used to address the flow statement table and the current statement frequency count table. If either of these is set to zero, the correpsonding option is not in effect. If both are set to zero, control is returned to compiled code.

If one or other of the options is in effect, there are four possible cases that require different action:

1. A branch-in following a branch-out or vice versa.
2. A branch-in following another branch-in
3. A branch-in to a new block.
4. Return from an on-unit to the point of interrupt.

These cases are dealt with individually in the sections that follow.

Case 1. Branch-In Following a Branch-Out or Vice Versa This situation indicates non-sequential flow of control, and must therefore be recorded in the FLOW and COUNT tables. For FLOW, the new statement number together with flags indicating a branch-in, a branch-out, or a branch-in to a procedure or on-unit, are entered in the position indicated by the pointer at the head of the flow statement table. The pointer is then updated to point to the next available space. If the next space would be outside the table, the pointer is reset to the head of the statement number section of the table.

For COUNT, the count value in the field for the appropriate statement number is altered. For a branch-in, the count of the statement branched to is incremented by one. For a branch-out, the count of the statement after the statement branched from is decremented by one.

If IBMBEFL is being called for a branch in, it is possible that it was caused by a GOTO-out-of-block and a new procedure or on-unit name may need to be recorded. In this situation, IBMBEFLC will have been called during the execution of the GOTO-out-of-block code and will have set a flag in the flow statement table. The flag is therefore tested and, if it is found on, the entry is treated as an entry to a new block. See case 3.

A further possibility is that the branch-in will be a return to a CALL statement or a function reference. These are distinguishable because the call to IBMBEFL is made by a BAL instruction rather than a BALR instruction. If the COUNT option is in effect, this must be tested for, and the count value of the next statement rather than the current statement be incremented This is necessary because the branch-out and the branch-in for CALL statements and function references are both made at the same statement, (see description under "Action during Compilation" earlier).

Case 2. A Branch-in Followed by Another Branch-in: No action need be taken as such a situation can only be caused by sequential flow. For example consider the statements:

```
LAB1:  X=Y;
LAB2:   Z=X;
```

Both LAB1 and LAB2 are potential branch-in points, but, if a call to IBMBEFL is made for LAB2 immediately after a call has been made for LAB1, it is plain that the flow of control has been sequential. Consequently when a branch-in follows another branch-in, IBMBEFL returns control to compiled code without taking any action.

This situation does not arise with branch-out points, because the code to call IBMBEFL is only executed if the branch is taken.

Case 3. A Branch-In to a New Block: This case requires that block information be entered for the FLOW option, and that, for the COUNT option, a check be made to see whether a new external procedure has been entered. If it has, a different statement frequency count table will have to be used because there is one for each external procedure.

Special action will be required if the block entered is an on-unit. This is because the branch-out will have been made at the point of interrupt and this will not have been automatically recorded by a call to IBMBEFL. When a new block is entered a test is therefore made on the DSA flags of the block to establish whether it is an on-unit. The action taken if it is a on-unit is described later under the heading "Branch-In to an On-Unit."

After any action required to handle entry into an on-unit, the following will take place.

For FLOW, the name of the block must be discovered and placed in the next available space in the names section of the flow statement table. Also, the statement number entry must be flagged to show that it marks a change of block. The procedure name is found following the DSA chain back until a procedure DSA is found and accessing the name, which is held at a standard offset from the entry point of the procedure. When the procedure name has been found, the statement number and flags, and the procedure name, are placed in the appropriate sections of the flow statement table and the pointers altered to point to the next available fields.

For COUNT, a check must be made to discover whether a new statement frequency count table is required. This is done by comparing the address in the register 3 save area of the DSA of of the procedure that called IBMBEFL with that at offset X'4' in the current statement frequency count table. If they are the same no action is required, because the new block must have the same static control section as the previous block and consequently must be in the same external procedure. If the addresses are not the same, a search is made down the chain of statement frequency count tables for a matching table. If one is found, the address of the current table is set to point to the table that has been found, and the required entry made in that table. If no matching table is found, a new table must be set up.

Creating a New Statement Frequency Count Table: Before creating a new statement frequency count table, IBMBEFL checks to see if a statement number table exists for the new procedure. If it does not, counting will not take place. In this situation, the current statement frequency count table is flagged to indicate that counting is to be suspended until another procedure is entered, and control is returned to compiled code.

Provided a statement number table does exist, a new statement frequency count table will be required. IBMBEFL first obtains the required amount of non-LIFO storage for the table. One fullword is required for every statement in the external procedure if it was compiled with the GOSTMT option, and two fullwords are required for every statement if it was compiled with the GONUMBER option. The count fields are set to zero, and, for procedures compiled with the GONUMBER option the numbers are inserted in the tables. The new table is then linked with its matching external procedure by placing the address of the static control section for the procedure in the new table.

Branch-in to an On-unit: If the code that called IBMBEFL is found to be in an on-unit, special action is required. The statement number for the point of interrupt must be discovered and appropriate entries made in the flow and count tables, before the data for the entry to the on-unit can be recorded. This is because there will have been no call to IBMBEFL at the point of interrupt to register a branch-out. The statement number of the interrupt is found by IBMBEFL in the same way as that used by the error message modules, described earlier in this chapter. When the number has been found, it is incorporated in the flow and count tables as if it were a normal branch-out. The branch-in entry is then handled as if it were a normal entry to a new block. It is possible for the FLOW option to be in effect without there being a statement number table available. In this situation, a statement number of zero is entered in the flow statement table for the branch-out at the point of interrupt.

A problem also exists for COUNT if an interrupt results in the termination of a program. In this situation, the interrupt point must be marked as a branch/out, otherwise, statements after the interrupt would have an incorrect count value. This situation is checked for when the FINISH condition is raised. During the handling of the FINISH condition, the GOTO code is executed and IBMBEFL is called at entry point C. A check is then made to see if FINISH was raised because of an interrupt. If it was, the point of interrupt is discovered and entered as a branch-out point in the appropriate statement frequency count table.

Case 4. Return from On-unit to Point of Interrupt: When return is made from the end of an on-unit to the statement that caused the interrupt, there will be nc automatic call (resulting from code inserted during compilation) to IBMBEFL. The necessary information for the flow and statement frequency count tables is therefore entered when IBMBEFL is called at the end of the on-unit. The statement numbers passed for such calls are specially flagged so that IBMBEFL discovers the point of interrupt and takes the necessary action to update the flow statement table and statement frequency count tables.

Action on Output

Interpreting the Flow Statement Table

Information from the flow statement table is interpreted by the message module IBMBESN or the PLIDUMP routines, and transmitted in the form of statement number pairs which are associated with the names of procedures or with on-unit condition types.

To extract the information, the message module must know from which points output in the statement number and procedure names section of the table output is to start. It must also be able to match the entries in the two sections of the table.

The starting points in both sections of the table are found by checking whether the dummy entry, inserted during program initialization, has been overwritten. If the dummy entry has not been overwritten,

|the starting point is the first entry in |that section of the table. If the dummy |entry has been overwritten, the starting |point will be the entry flagged as the next |available entry. This is because the table |is used cyclically, with the newest entry |overwriting the oldest entry.
|
| Statement numbers are matched with |procedure names by comparing the number of |procedure names with the number of |statement number entries that are flagged |as being associated with procedure name |entries. If the two numbers are the same, |the first procedure name will be associated |with the first statement number that |requires a procedure name. If there are |more procedure names than statement numbers |that require procedure names, the trace of |procedures must be longer than the trace of |statement numbers. Accordingly, the |procedure names are put out without |statement numbers until the point is |reached where the number of procedure names |left is the same as the number of statement |numbers that require them. From that point |on statement numbers and procedure names |are put out together. If there are more |statement numbers that require procedure |names than there are procedure names, the |trace of statement numbers must be longer |than the trace of procedure names. The |earliest statement numbers are put out |without names and, where a procedure name |is required, "UNKNOWN" is used. When the |number of names required matches the number |available, the procedure names are put out |with the statement numbers.
|

|Interpreting the statement frequency count |tables
|
|Module IBMBEFCA is called at program |termination to print count information. |Output is tabular and printed three columns |to a page. An entire page is built before |transmission.
|
| Output for a procedure begins with the |procedure name. This is followed by the |column headings: "FROM TO COUNT". The |current count is initialized to zero and |the first non-zero entry in the table is |found. The associated statement number is |then placed in the 'FROM' part of a |temporary line and the value for the non-|zero entry is added to the current count. |The entries for the following statements |are scanned until one with a non-zero count |value is found. The number of the |preceding statement is then placed in the |'TO' part of the line and the current count |in the 'COUNT' part. This line is included |in the page. The statement number found is |then placed in the 'FROM' part of the |temporary line and its branch count (which |may be negative) is added to the current |count. The scan of entries continues until |another non-zero count is reached, and the |process is repeated.
|
| If the count for a range is zero, the |line is not moved into the page but the two |statement numbers are saved for separate |printing. Whenever a line is moved into |the page, checks are made for the end of a |column and the end of the page. When the |page is full it is transmitted.
|
|The process is continued until the end of |the table is reached.
|
|The next table is then processed, until all |procedures have been handled.
|
|Finally, ranges of unexecuted statements |are printed for each procedure.

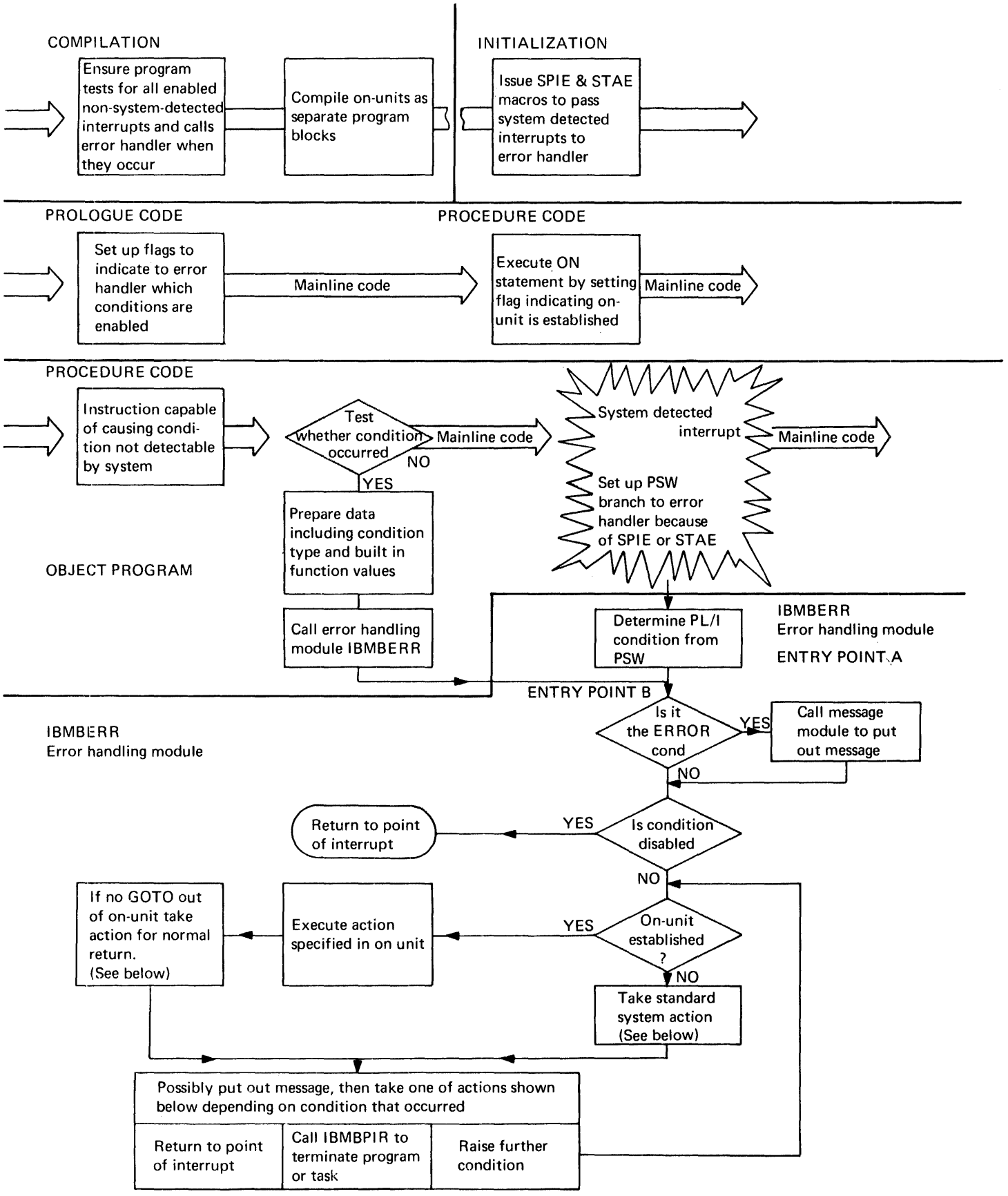


Figure 7.15. Outline of error handling

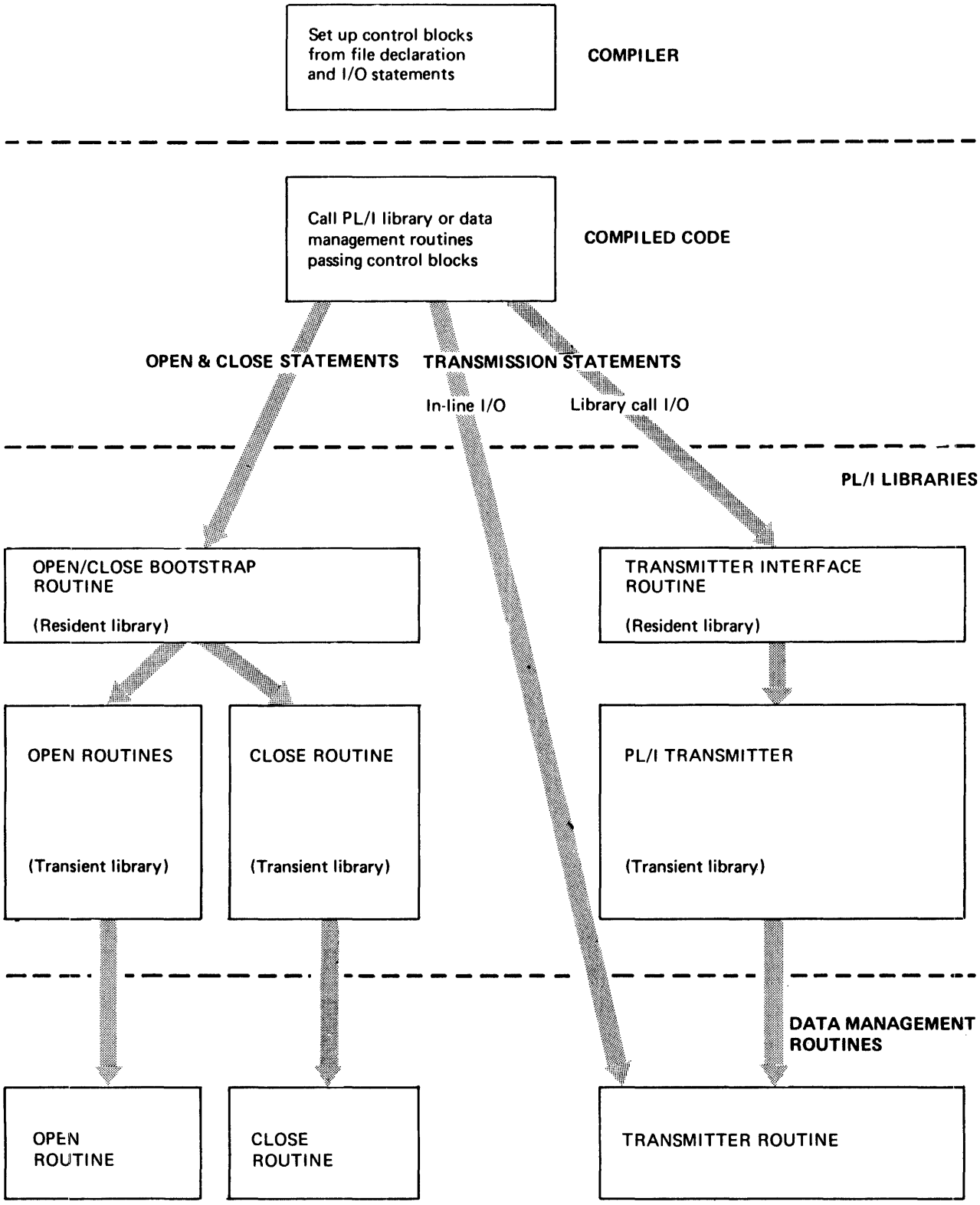


Figure 8.1. The principles used in record I/O implementation

Chapter 8: Record-oriented Input/Output

Introduction

This chapter considers the implementation of the following statements:

> File declarations
> Open and close statements
> READ, WRITE, DELETE, LOCATE, UNLOCK, and REWRITE statements referred to generically as transmission statements

Together, these statements make up record I/O.

The OS PL/I Optimizing Compiler uses the data management routines of OS/360 to implement record I/O. These routines offer facilities similar but not identical to those of the PL/I language. The data management routines require that:

1. A data control block (DCB) is set up to describe and identify the data set.
2. OPEN and CLOSE macro instructions are issued to open and close the data set.
3. GET, PUT, READ, or WRITE macro instructions are normally issued to store or obtain a new record.

The data management routines transmit the data one block at a time between the data management buffer and the external medium, but each separate macro instruction issued by the program results in only a single record being passed. When a transmission error occurs, or when the end-of-file is reached, the data management routines either set flags indicating the error or branch to error-handling or end-of-file routines that can be specified by the programmer.

The basic method used by the optimizing compiler to implement record I/O is to retain the source program information in a number of control blocks, and to pass these control blocks to PL/I library routines which interpret the information and carry out the necessary action by calling data management routines in the appropriate manner. The method is summarized below, and shown diagramatically in figure 8.1. Figure 8.15 shows the overall scheme in greater detail.

Summary of Record I/O Implementation

File Declarations

For a file declaration, the compiler generates two control blocks: the declare control block (DCLCB) and the environment control block (ENVB). Together, these two control blocks contain a complete record of the file declaration.

OPEN Statements

OPEN statements are compiled as a call to a resident-library bootstrap routine, IBMBOCL, which has passed to it an open control block (OCB) containing the attributes and environment options that have been used in the OPEN statement.

The bootstrap routine loads and calls a number of transient routines that build a definitive control block, known as the file control block (FCB), from information in the DCLCB, ENVB, and OCB. The file is associated with the data set, and the appropriate PL/I transmitter module is loaded.

The FCB is used during the execution of transmission statements to access all file information. It is addressed via the DCLCB and the pseudo-register vector.

Transmission Statements

For the majority of file and statement types, details of statement type, of record, key, and event variables are set up in control blocks during compilation; during execution, these control blocks are passed to a resident-library interface routine, IBMBRIO. IBMBRIO then calls a PL/I transient-library transmitter module, which issues the appropriate data management macro instruction, and checks for errors, before returning control to compiled code. This method is known as library-call I/O.

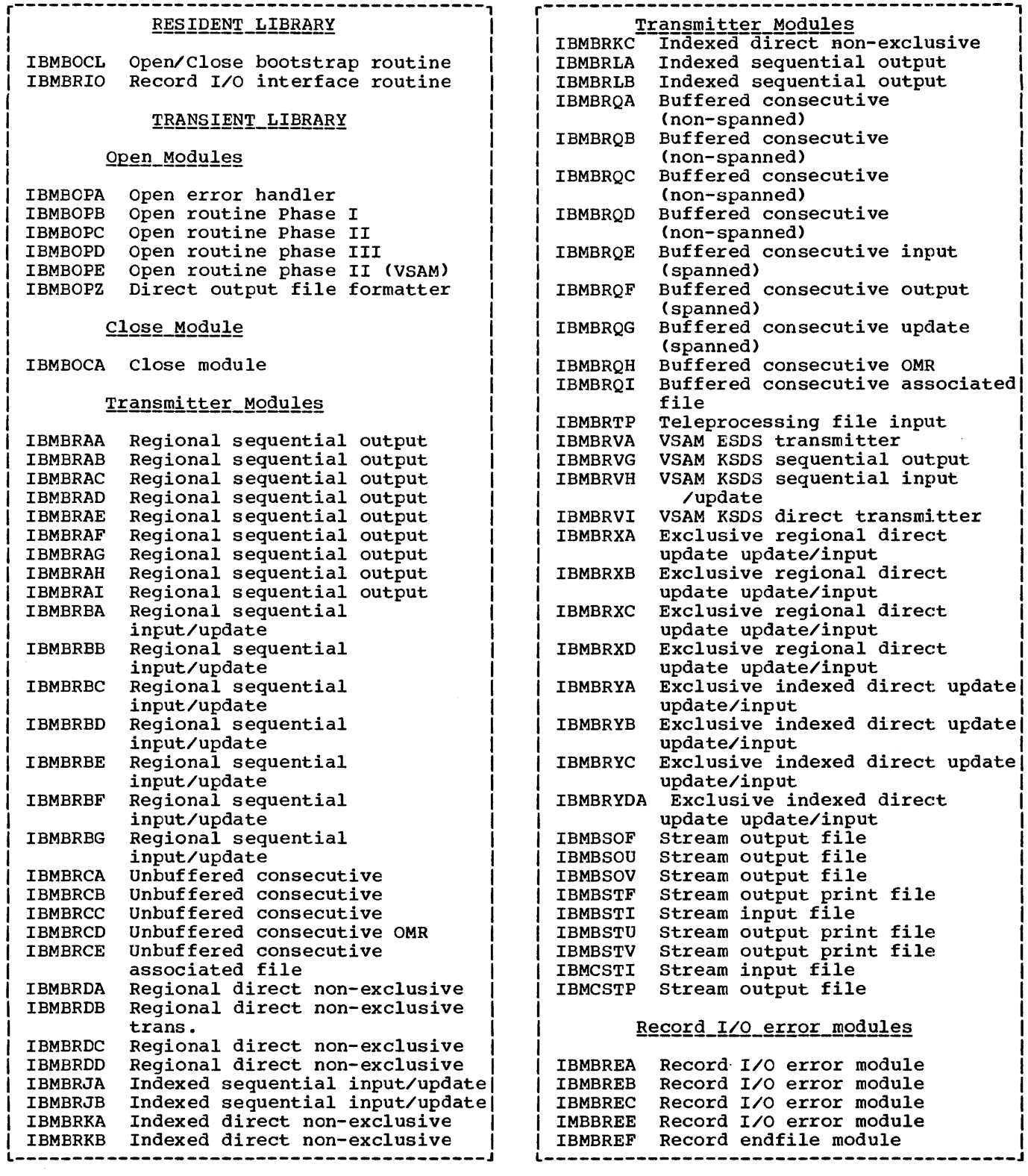


Figure 8.2. Library subroutines used in record I/O

If the TOTAL option is used, the majority of transmission statements on buffered consecutive files are compiled as direct calls to the data management routines. This method is known as in-line I/O. When using in-line I/O, subroutines of the PL/I transmitters are available to handle error situations. The appropriate

FILE TYPE	ACCESS METHOD
Buffered consecutive	QSAM/VSAM
Unbuffered consecutive	BSAM/VSAM
Regional sequential (not spanned records)	BSAM
Regional sequential (spanned records only)	BDAM
Regional direct	BDAM
Indexed sequential	QISAM/VSAM
Indexed direct	BISAM/VSAM
TP buffered input/update	TCAM
VSAM	VSAM

Consecutive or indexed files can be used to access VSAM data sets; the PL/I open routines will determine the data type. For details see section on OPEN statement.

Figure 8.3. Access methods and file types

transmitter is loaded during the file opening process.

CLOSE Statements

CLOSE statements are implemented by a call to the open/close bootstrap routine IBMBOCL, which loads and calls the transient close routine IBMBOCA. This routine disassociates the file from the data set, and handles the necessary housekeeping.

Implicit Open

Implicit opening is handled by manipulation of addresses so that any attempt to access the file when it is not open will result in control being passed to the open routines in the PL/I libraries.

Implicit Close

Implicit closing is handled by the program termination routine checking for open files, and if it finds any, calling the PL/I library routine to close them.

As can be seen from the summary above, a large number of library subroutines and control blocks are used in the implementation of record I/O. These are summarized in two figures: figure 8.2 for library subroutines and figure 8.4 for control blocks. More detailed descriptions for each statement type are given below.

ACCESS METHOD

The access method used for different PL/I file types is shown in figure 8.3.

File Declaration Statements

For each file declaration, a declare control block (DCLCB) and, optionally, an environment control block (ENVB) are set up. Both are held in static internal storage for internal files, or in a separate control section for external files.

The DCLCB is a control block that contains the filename together with a record of the attributes obtainable from the file declaration, both those given explicitly and those deducible by default. This information is retained until the file is opened, when, unless the TOTAL option has been used in the file declaration, the information is merged with any attributes in the OPEN statement.

The ENVB contains the addresses of all environment options. The format of the ENVB is given in appendix B.

From information in the DCLCB and the ENVB, (and sometimes from the open control block (OCB) produced from the OPEN statement) a further control block, the file control block (FCB) is generated. During execution of an I/O statement, all information about the file is derived from the FCB.

Execution

No executable code is produced from the file declaration. Figure 8.5 shows the code resulting from a file declaration.

CONTROL BLOCKS GENERATED FROM FILE DECLARATION

DCLCB

Function Holds all file attributes used in file declaration
Location Separate control section for external files, static internal for internal files.
When generated: During compilation
Contents
- Record of file attributes at declaration
- File name
- Address of ENVB
- Offset of FCB pointer in PRV

Environment control block (ENVB)

Function: Holds information on environment options
Location: In static storage
When generated: During compilation
Contents: Addresses of
- blocksize
- record length
- of buffer tracks
- KEYLOC value
- key length
- indexarea size
- addbuf

CONTROL BLOCK GENERATED FROM OPEN STATEMENT

Open control block (OCB)

Function: To contain file attributes given in OPEN statement
Location: In static storage
When generated: During compilation
Contents: The attributes when specified on the OPEN statement

Figure 8.4. (Part 1 of 2). The fields used in implementing record I/O

CONTROL BLOCKS GENERATED FROM INPUT/OUTPUT STATEMENTS

Key descriptor (KD)

Function: To describe the key variable
Location: Depends on storage class of key variable
When generated: Depends on storage class of key variable
Contents: Length and address of key variable

Record descriptor (RD)

Function: To describe the record variable
Location: Depends on storage class of record variable
When generated: Depends on storage class of record variable
Contents: Length and address of record variable

Request control block (RCB)

Function: Holds a definition of the statement for execution-time checking
Location: In statis storage
When generated: During compilation, for library data management calls only
Contents: Flags defining statement
Code for TM instruction, or a branch instruction (if checking was done during execution)

CONTROL BLOCK GENERATED DURING EXECUTION OF OPEN STATEMENT

File control block (FCB)

Function: Acts as a central source of information about the file
Location: In statis storage
When Generated: During open
Contents include:
Flags indicating valid statements
Transmitter name
Transmitter address
Error module address
DCB/ACB address
Filename address
Buffer address
flags and workspace for the Xmitters
DCB
Data Management control block/ Access-Method Control Block

Figure 8.4. (Part 2 of 2). The fields used in implementing record I/O

OPEN Statement

Compiler Output

For an OPEN statement, the compiler generates a call to the open/close bootstrap routine, IBMBOCL, and an open control block (OCB). The OCB holds any attributes that are declared in the OPEN statement.

More than one file may be passed to the open routines. The last file has its last parameter flagged with its first bit set to '1'.

Execution

For an explicit open, a call is made to the open/close bootstrap routine, IBMBOCL. For each file to be opened, the following information is passed to IBMBOCL:

The address of the DCLCB
The address of the OCB, (or zero, if no OCB exists)
The address of the TITLE, (or zero if none is specified)

IBMBOCL has four entry points:

IBMBOCLA explicit open
IBMBOCLB implicit open for library call I/O
IBMBOCLC explicit close
IBMBOCLD implicit close

When called by entry point A, IBMBOCL invokes the transient library open routines to open the file. If the environment option TOTAL has not been used in the file declaration, it will be necessary to determine the attributes of the file by merging the attributes in the file declaration with those used in the OPEN statement. Attributes in the file declaration are held in the ENVB and DCLCB. Attributes used in the OPEN statement are held in the OCB. If the TOTAL option has been used, attributes are taken from the declaration, and any contradictory attributes in the OPEN statement result in the raising of the ERROR condition.

The open modules build an FCB and DCB from the information in the control blocks, initialize the pseudo-register vector to point to the FCB, load the PL/I and data management transmitters, and return to compiled code. Five transient open modules are used. Their functions are summarized below and are described in detail in the licensed publication OS/360 PL/I Transient Library: Program Logic.

Actions Carried Out by Transient Open Routines

The transient open routines perform the following major functions when opening a file:

1. Build the file control block (FCB) and data control block (DCB), or, for VSAM the access method control block (ACB) for the file. The FCB is a PL/I control block used to access all file information. The DCB is a data management control block used to describe the data set. The ACB is the equivalent of the DCB for VSAM files.
2. Issue the data management OPEN macro instruction to associate the file with the data set.
3. Obtain and initialize buffers and any other blocks required for the file.
4. Determine which statement types are valid for the file, and store this information as a set of flags held in the FCB.
5. Select the appropriate PL/I transmitter, and load it for use during transmission statements.
6. Check for errors, and raise the UNDEFINEDFILE condition if any are found.
7. Place the address of the FCB in the correct pseudo-register vector offset.

The execution of an OPEN statement is summarized in figure 8.6.

VSAM Data Sets

VSAM data sets both KSDS and ESDS are normally accessed by PL/I using VSAM macro instructions, however, in certain circumstances the data sets are accessed through the compatibility interface. If the file is declared with ENV (VSAM) the VSAM macro instructions will automatically be used. Even if it is not so declared, the PL/I open modules will normally detect that a VSAM data set is being accessed. To do this they issue an RDJFCB macro instruction. However this action is not effective if the ALLOCATE command is being used under TSO to provide DD information,

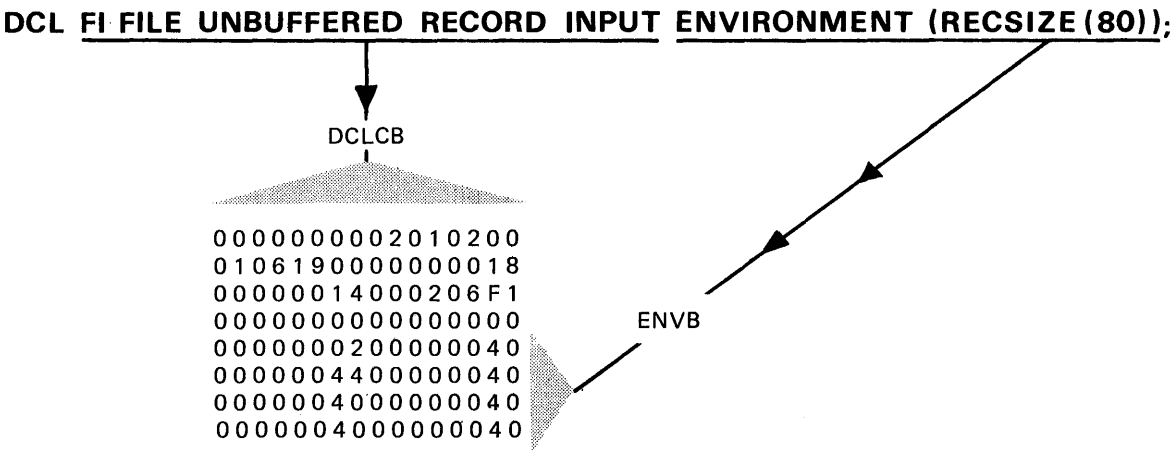
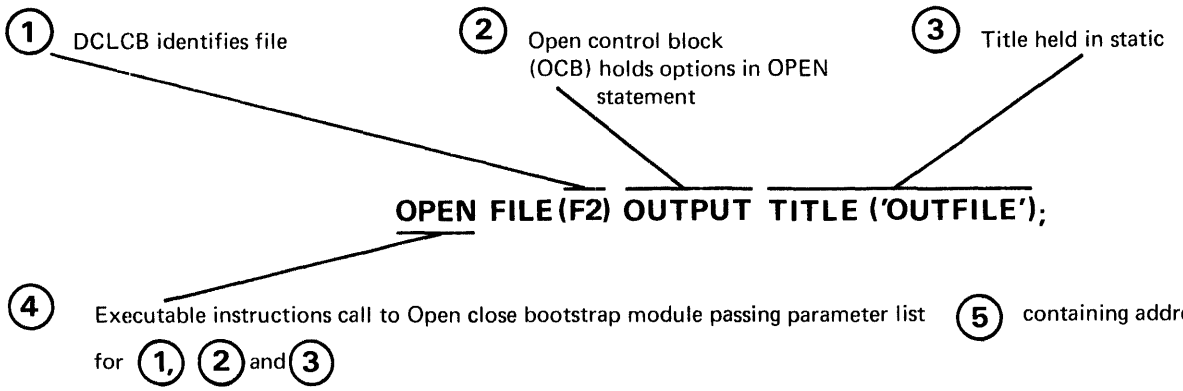


Figure 8.5. Information in the file declaration is held in the ENVB and the DCLCB until the file is opened



(1) DCLCB set up during file declaration see figure 8.5

(2) Open control block in static. See Appendix A for Format :

```
000048  0020000000D00800 CONSTANT
        00000000
```

(3) Title (held in static internal) is addressed via locator (also in static internal)

Title

```
0000A0  D6E4E3C6C9D3C5
```

Locator

```
000020  000000A000070000
```

(4) Machine Instructions

```
000088 41 10 3 064  LA    1,100(0,3)        } Point R1 at P-lists
00008C 58 F0 3 00C  L     15,A..IBMBOCLA    } Branch to open/close bootstrap
000090 05 EF        BALR  14, 15            }
```

(5) Parameter list

```
000064 00000044   A..CONSTANT          No. of files to be opened
000068 00000000   A..DCLCB
00006C 00000048   A..CONSTANT          A...OCB
000070 00000020   A..CONSTANT          A...LOCATOR for TITLE
000074 00000000   A..NULL ARGUMENT     } Used for print files only
000078 80000000   A..NULL ARGUMENT     }
```

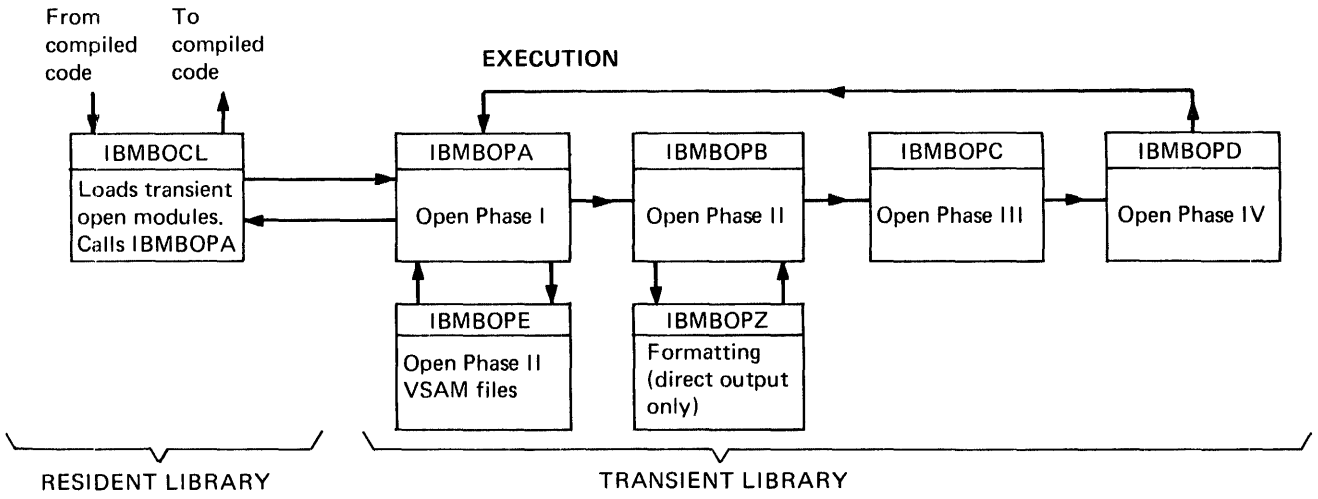


Figure 8.6. Open statement

because, in this case, the RDJFCB macro instruction cannot determine that a VSAM data set is being accessed. In this situation the compatibility interface will be used. It is possible for the user to force the use of the compatibility interface by specifying either "RECFM" or "OPTCD=L" in the AMP parameter of the DD statement.

The flow through the PL/I open modules is as follows. IBMBOPA scans the list of files to be opened and sets a flag to indicate that IBMBOPE is required for any files declared with ENV (VSAM). If one or more files are found without ENV (VSAM), IBMBOPB is called to open them. Then on return from IBMBOPB, IBMBOPE is called to open any VSAM files. If IBMBOPB detects that any consecutive or indexed files are being used to access VSAM data sets, it will set the flag indicating that IBMBOPE is required and ignore that file. When all the non-VSAM files have been opened, IBMBOPD returns to IBMBOPA. IBMBOPA tests to see whether there are any VSAM files to be opened, and, if ther are, calls IBMBOPE.

IBMBOPE opens the files starting with the first. Each file is completely opened before starting to process the next. The open process involves nine main steps, as follows:

1. Merge attributes from OPEN statement with file declaration and check for validity.

2. Get non-LIFO storage space for the FCB and ACB, and create the ACB using the GENCB macro instruction. The DDNAME is obtained from the filename or the TITLE option. The password is obtained from the PASSWORD environment option if specified. The MACRF options used are:

SEQ/DIR	SEQUENTIAL/DIRECT
KEY/ADR	KSDS/ESDS
IN/OUT	INPUT/OUTPUT (both specified for UPDATE)

3. Issue an OPEN macro instruction and test the return codes in the ACB.

4. Check the actual values of the RECSIZE, KEYLENGTH, and KEYLOC options against any values specified in the ENVIRONMENT option. Check that NCP/STRNO is not greter than one. If any errors or descrepancies are found, the ACB must be closed.

5. Set up the mask of invalid statements for use by IBMBRIO.

6. Get non-LIFO storage space for the IOCB and RPL, plus key space for a KSDS, and a dummy buffer for a buffered file. Create the RPL using a GENCB macro instruction. The OPTCD values are partially set as shown below. The transmitter merges the other options according to statement type. The OPTCD options set are:

KEY/ADR	KSDS/ESDS
SEQ/DIR	SEQUENTIAL/DIRECT
UPD/NUP	UPDATE/INPUT or OUTPUT
GEN/FKS	GENKEY/not GENKEY

KEQ,MVE, and ASY are always specified.

7. Load the approptriate library transmitter as follows:

ESDS	IBMBRVAA
KSDS SEQUENTIAL OUTPUT	IBMBRVGA
KSDS SEQUENTIAL INPUT/UPDATE	IBMBRVHA
KSDS DIRECT	IBMBRVIA

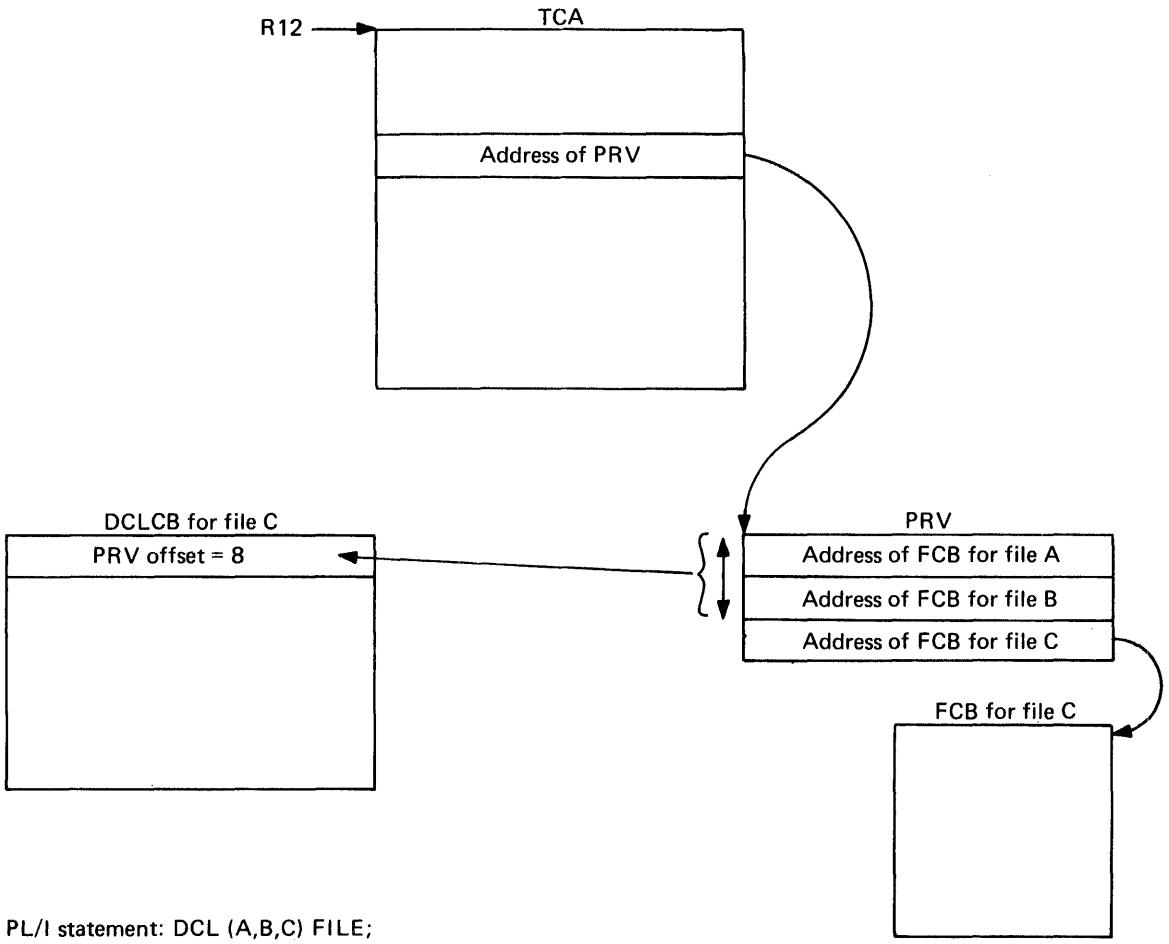
8. Insert "E" as the seventh character of the error module name, so that IBMBREEA will be loaded if an error occurs.

9. Add the FCB address to the chain of open files and set the address of the FCB in the pseudo register.

The FCB and File Addressing

During execution of record I/O statements, all information about the file is obtained from the FCB. However, as the FCB is nct created until execution, the FCB cannot be addressed directly by compiled code. Instead, compiled code obtains from the DCLCB the offset within the PRV at which the FCB address is held. This offset is placed in the DCLCB by the linkage editor. The mechanism is illustrated in figure 8.7.

The use of the pseudo-register vector allows separately compiled programs to refer to the same FCB for an external file, even though the address of the FCB cannct be known until execution. An explanation of the use of the pseudo-register vector is given in chapter 2, under the heading "Use of the Pseudo-Register Vector."



PL/I statement: DCL (A,B,C) FILE;

The address of the FCB for the file is obtained by adding the offset in the DCLCB to the PRV address which is held in the TCA

Figure 8.7. Addressing files via DCLCB and PRV

Transmission Statements (Library-Call I/O)

Compiler Output

For transmission statements the compiler generates a call to the PL/I transmitter interface module, IBMBRIO. IBMBRIO has the following parameter list passed to it:

Address of DCLCB
Address of request control block (RCB)
Address of record descriptor (RD); or,
  address ignore factor; or,
  address at which to set pointer
  Address of key descriptor (KD); or,
    zero if no key descriptor
  Address of event variable (EV); or,
    zero if no event variable
  Abnormal locate return address (LOCATE statements only)

The DCLCB is generated from the file declaration, as described earlier in the chapter. The remainder of the control blocks in the parameter list are generated for the transmission statement.

The request control block (RCB) defines the statement type. It consists of two words. The first is a fullword of flags that define the statement type and option, indicating whether the statement is READ SET, READ INTO, WRITE FROM, etc. The second word is a test-under-mask (TM) instruction that is executed by IBMBRIO to check whether the statement is valid. The flags in the RCB are tested against flags in the FCB or dummy FCB. If the statement is invalid, a branch is made to an address held in either the FCB or the dummy FCB. If the file is not open, the dummy FCB will be accessed, and the branch will be made to the open/close bootstrap to open the file. If the file is open, a real FCB will be accessed, and the branch will be via a bootstrap to the error handler. The RCB is set up in static internal storage. The format is shown in appendix B.

The record descriptor (RD) contains the address, length and type of the record variable. (The record variable is the variable to or from which the record will be transmitted.) A record descriptor is generated only if a record variable is used. The format is shown in appendix B.

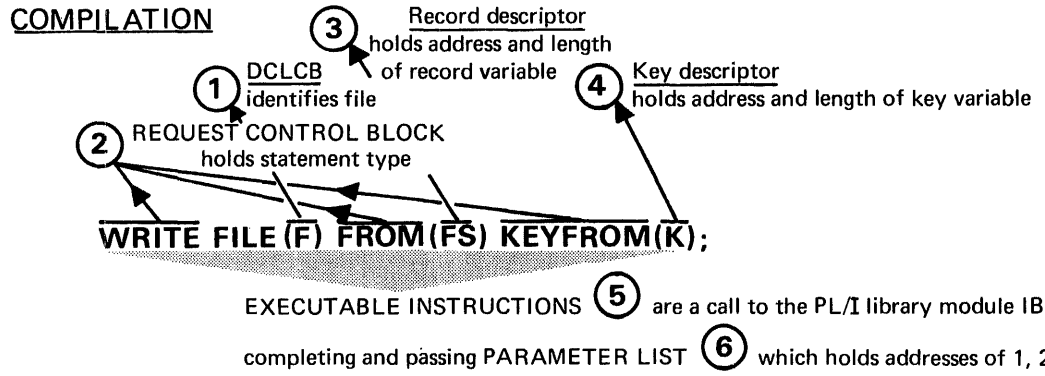
The key descriptor (KD) contains the address and length of the key variable. (The key variable is the variable to or from which the key will be transmitted.) It is generated only if a key variable is used. The format is shown in appendix B.

If the record variable or the key variable is STATIC INTERNAL, a complete RD or KD is set up and placed in static internal storage during compilation. In most other circumstances, a skeleton RD or KD will be set up, and will be completed by the inclusion of the address during execution. The completed descriptor may be moved into temporary storage. In certain conditions, no skeleton is produced; instead, the complete descriptor is built in temporary storage by compiled code.

The event variable (EV) (if used) contains information about the event that has been associated with the event I/O statement. (For format, see appendix B.) The implementation of event I/O is covered briefly at the end of this chapter, and further in chapter 11 for non-multitasking programs and chapter 14 for multitasking programs.

The abnormal locate return block is used only for LOCATE statements. It is the address of a block containing the address to which control will be passed if an error is detected in a LOCATE statement and a normal return is made after execution of the on-unit. The abnormal-locate return address is usually the start of the next statement.

The code and control blocks generated for a transmission statement using a library call to the data management routines are shown in figure 8.8.



---

(1) DCLCB, set up from file declaration holds address of FCB via pseudo register vector. (See file declaration).

(2) REQUEST CONTROL BLOCK holds record of statement type

```
000028 0880200091022001 CONSTANT
```

(3) RECORD DESCRIPTOR holds address and length of record, set up as far as possible during compilation, completed during execution. For statement above set up in temporary storage during prologue code

(4) KEY DESCRIPTOR holds address and length of key, set up as far as possible during compilation, but, for this statement, completely built by compiled code in temporary storage (see 5).

(5) Executable instruction

```
* STATEMENT NUMBER 4
000092  41  90  D  0B8     LA    9,184(0,13)      Pick up address record descriptor
000096  50  90  3  084     ST    9,132(0,3)       Place in parameter list
00009A  41  90  D  0B0     LA    9,176(0,13)      Pick up address-key descriptor
00009E  50  90  3  088     ST    9,136(0,3)       Place in parameter list
0000A2  41  10  3  07C     LA    1,124(0,3)       Point R1 at parameter list
0000A6  58  F0  3  014     L     15,A..IBMBRIOA
0000AA  05  EF             BALR  14,15            Call IBMBRIO
```

Note: For this statement the record and key descriptors were set up in temporary storage during prologue code.

(6) PARAMETER LIST passed to IBMBRIO

```
00007C  00000000   A..DCLCB            Filled in by linkage editor
000080  00000028   A..CONSTANT         Request control block
000084  00000000   A..RD               (Record descriptor)
000088  00000000   A..KD               (Key descriptor (built during execution))
00008C  00000000   A..NULL ARGUMENT
000090  80000000   A..NULL ARGUMENT
```

Figure 8.8. (Part 1 of 2). Handling a transmission statement

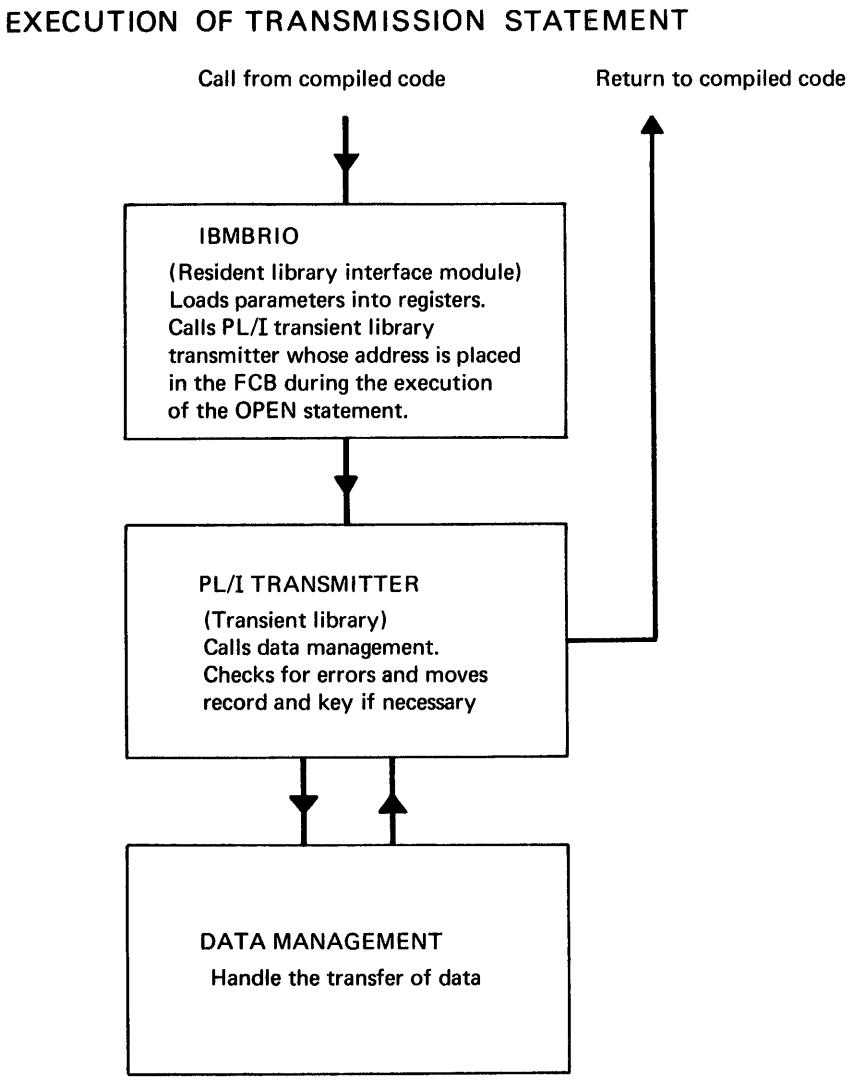


Figure 8.8. (Part 2 of 2). Handling a transmission statement

Execution

Compiled code calls the transmitter interface module, IBMBRIO, passing to it the parameter list shown above under "Compiler Output".

The interface module, IBMBRIO, first acquires a DSA, which is used by IBMBRIO and by the transmitter. It then initializes the registers for the transmitter, and executes the TM instruction in the request control block (RCB). This instruction tests a set of flags that are addressed by a pseudo-register offset contained in the DCLCB. The contents of the pseudo-register offset depends on whether the file is open. If the file is not open, it is opened and return made to this point to continue the statement. (See "IMPLICIT OPEN" later in this chapter.)

When the file is open, the TM instruction tests the validity flags in the FCB. This establishes the validity of the statement. If the statement is not valid, a branch is made to the address held in the word in the FCB following the statement validity flags. This address is an entry point in IBMBRIO that calls the error handling module, IBMBERR, with an error code indicating an invalid statement.

If the statement is valid, a branch is made to the transmitter whose address is held in the FCB.

Transmitter Action

After the file is open and the statement validated, control is passed to the transmitter, which checks the record and key variables for errors, and issues the appropriate data management macro instruction. After the data management macro instruction has been executed, control returns to the transmitter. The transmitter moves the data between the data management buffer and the record variable, or sets the pointer to the record, and checks to see whether any errors have occurred.

Transmitter modules do not acquire separate DSAs, but use the DSA acquired by IBMBRIO.

If the statement is valid, control is returned to compiled code. The situation when an error has been detected is described later in this chapter under the heading "ERROR CONDITIONS IN TRANSMISSION STATEMENTS."

In certain conditions, data management will require a parameter list known as the data event control block (DECB). The PL/I library routines include this block in a PL/I control block known as the input/output control block (IOCB). A number of IOCBs may be used. The number depends on the file type, and on the NCP subparameter in the DD statement or NCP option in the ENVIRONMENT attribute. Depending on the file type, IOCBs may be generated during the execution of the open statement, or by the transmitters when they are required.

The format of the IOCB is shown in appendix A. The format of the DECB and a further description of its use is given in the publication OS/360 Supervisor and Data Management Macro Instructions. IOCBs are further described in the section "EVENT OPTION", below.

EVENT Option

When the EVENT option is used, transmission statements are always handled by library call. The compiler generates a call to IBMBRIO in the usual manner, except that the address of an event variable is passed in the parameter list.

The associated WAIT statement is compiled as a call to one of the library wait modules. The module called depends on whether or not the program is multitasking. The execution of an I/O statement with the EVENT option and its associated WAIT statement is shown in figure 8.9.

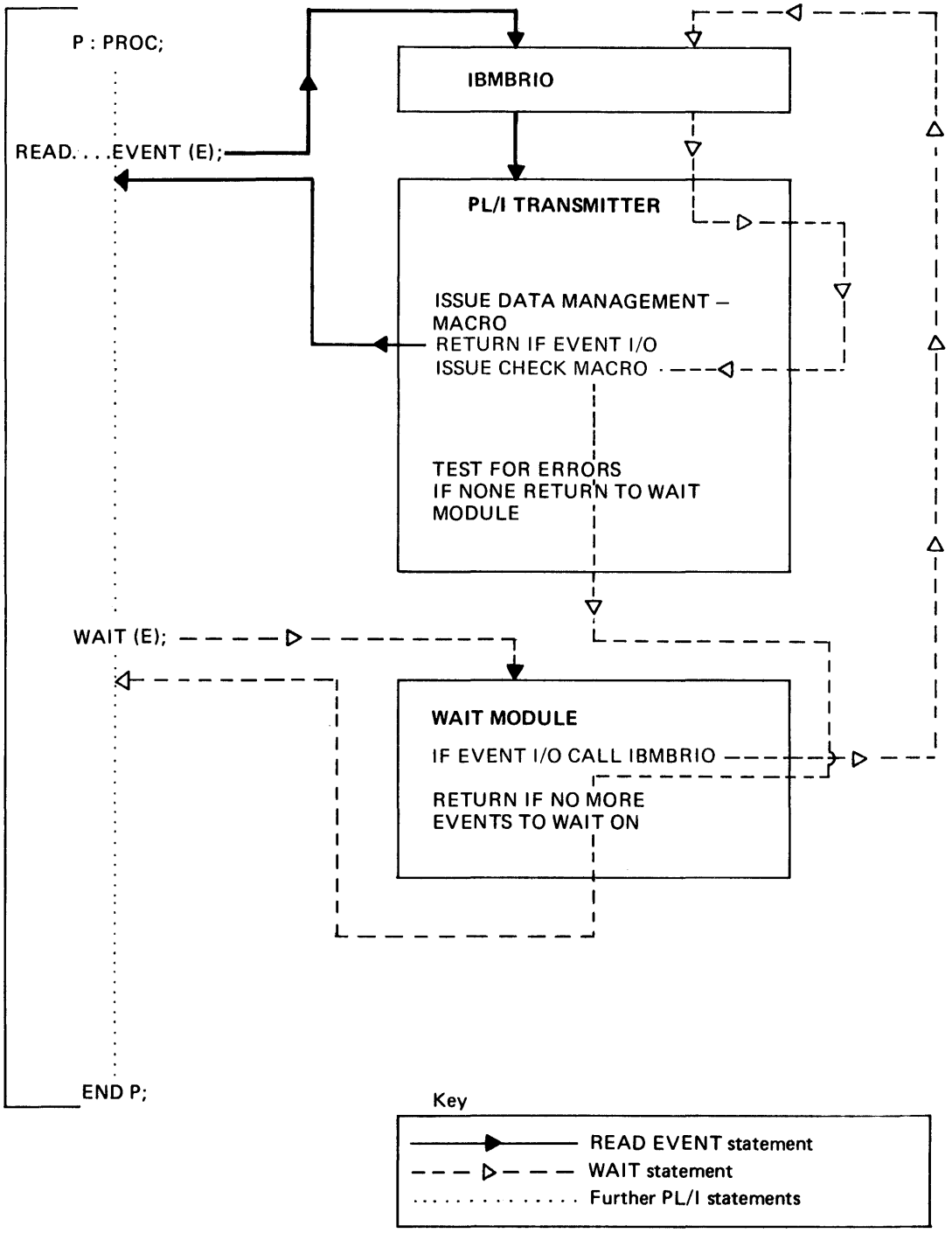


Figure 8.9. Handling the EVENT option

Execution

The principle used in event I/O is that the PL/I transmitter returns to compiled code as soon as the data management macro instruction has initiated the I/O.

When I/O with the EVENT option is being executed, the event variable associated with the event is set active and flagged to indicate that the event is an I/O event. When the WAIT statement is reached, the library wait module is entered. When the event is an I/O event, the PL/I library wait routine passes control to IBMBRIO. From information in the event variable, IBMBRIO locates the I/O operation associated with the event, and calls the transmitter. The transmitter then issues a CHECK macro instruction, and waits until the operation is complete. When control returns after the CHECK macro instruction, the transmitter assigns the transmitted data, and either returns to the wait module, or, if any errors are detected, enters one of the error routines. (For further details, see "ERROR CONDITIONS IN TRANSMISSION STATEMENTS" later in this chapter.)

When the transmitter assigns the data, it is necessary for the address and length of the record variable, and certain other information, to be available. This information is retained in the input/output control block (IOCB).

Use of the IOCB

The IOCB is chained to the event variable so that the I/O routines can access the statement when control is returned to them during execution of the WAIT statement.

To associate the PL/I statement with the data management operation, the DECB for the operation is included in the IOCB. (The DECB is a record held by the data management routines so that the operation can be posted complete.)

For certain types of PL/I files, the IOCB also contains the data management buffer to or from which the transmission will be made.

Allocation of IOCBs

For direct access files, IOCBs are allocated as they are required by the transmitter.

For sequential access files, the IOCBs are generated by the open routines. The number of IOCBs requested corresponds to the number specified in the NCP subparameter or option.

IOCBs and Dummy Records

In event I/O, the existence of a dummy record may not be discovered until after a read has commenced on the record following the dummy. When this happens, the DECB and IOCB pointers are reset appropriately.

Raising Conditions in Event I/O

Because the CHECK macro instruction is not issued until the WAIT statement is executed, PL/I conditions raised in event I/O are handled during execution of the WAIT statement. The implications of this are discussed in the section on the WAIT statement in chapter 11 for non-multitasking programs, and chapter 14 for multitasking programs.

Exclusive I/O

In exclusive I/O, records are protected from simultaneous updates from different tasks by use of the ENQ and DEQ macro instructions.

When a READ statement for an exclusive file is being executed, an ENQ macro instruction is issued. Unless NOLOCK is specified, the DEQ macro instruction is not issued until a REWRITE, DELETE, or UNLOCK statement is executed. For unblocked records, the ENQ and DEQ instructions are issued on one record only. For blocked records, they are issued on the data set.

Eight PL/I transmitter modules are used to handle exclusive files: they are shown in figure 8.2. The ENQ and DEQ macro instructions are issued by calling the resident library routine IBMBPDQ, which is addressed from the TCA.

The protection of the data set depends on all files that access the data set having the EXCLUSIVE attribute. If the data set is accessed by a file that does not have the EXCLUSIVE attribute, the data set will not be protected.

For VSAM files the EXCLUSIVE attribute is ignored and the NOLOCK option and UNLOCK

(1) DCLCB identifies file to be closed

CLOSE FILE(F2)

(2) Executable instructions consist of a call to the open/close bootstrap module passing parameter list (3)

---

(1) DCLCB set up for file declaration see figure 8.5

(2) Executable instructions

```
* STATEMENT NUMBER 5
0000AC 41 10 3 094      LA    1,148(0,3)          Place address DCLCB in p-list
0000B0 58 F0 3 010      L     15,A. .IBMBOCLC  }  Call open/close toolstrap
0000B4 05 EF            BALR  14,15            }
```

(3) Parameter list

```
000094  00000044   A. .CONSTANT      Address of constant showing number of files to be closed
000098  00000000   A. .DCLCB         Address DCLCB
00009C  80000000   A. .NULL ARGUMENT Used for disposition options, flagged in first bit to indicate last argument
```

CLOSE FILE(F1);

COMPILATION

```
L     7,F0          Pass address of constant with number of files to be closed
ST    7,2528(0,3)   Pass address of DCLCB of file
LA    1,2524(0,3)   Point R1 at parameter list
L     15,A. .IBMBOCLC  Branch to open/close bootstrap
BALR  14,15
```

EXECUTION

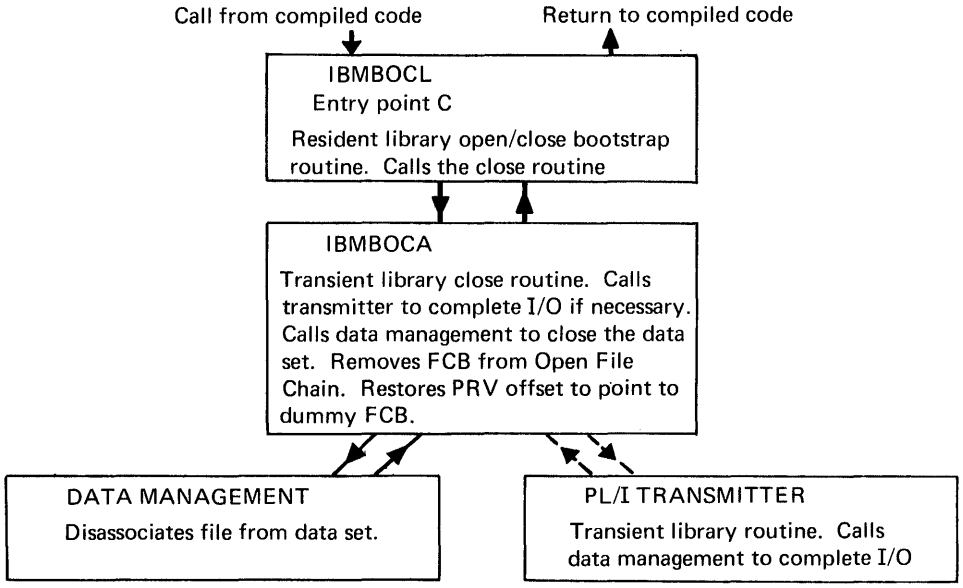


Figure 8.10. The execution of an explicit CLOSE statement

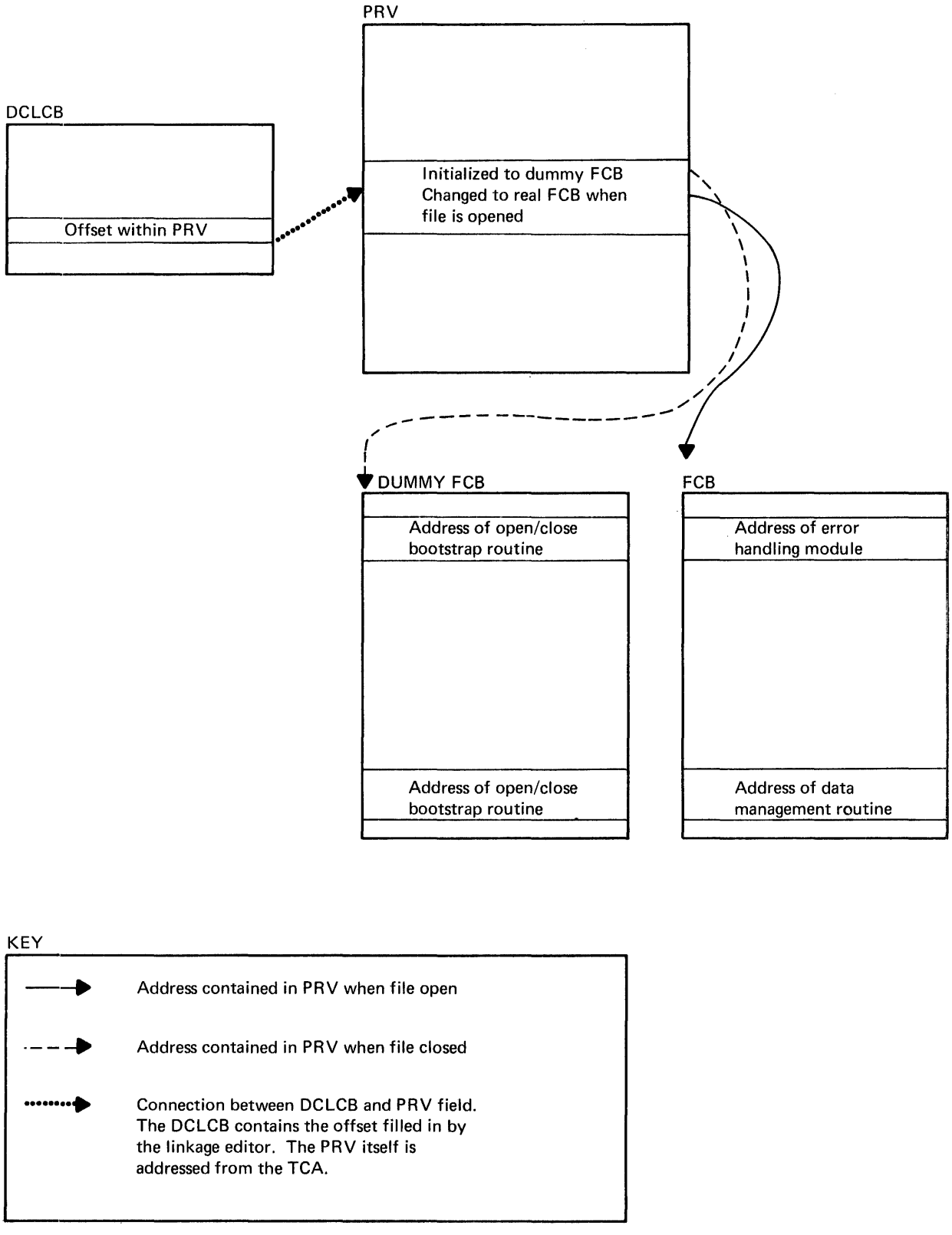


Figure 8.11. The addressing mechanism used during implicit open

|statement will have no effect (except that |for UNLOCK, the key specification is |checked.) Data set protection is provided by VSAM itself.

CLOSE Statements and Implicit Close

Compiler Output

For CLOSE statements, the compiler generates a call to the appropriate entry point of the open/close bootstrap module, passing it the addresses of the DCLCB and ENVB for the file.

No compiler action is taken for implicit close.

Execution

Files and data sets can be closed either by the PL/I CLOSE statement or by the termination of the program. In both cases, the close is carried out by library routines. The bootstrap module IBMBOCL is called either by compiled code, or, during program termination, by the termination routine, IBMBPIT or IBMTPJR for multitasking. It loads and calls the transient close routine, IBMBOCA.

The bootstrap routine IBMBOCL is passed a parameter list containing the addresses of the DCLCBs and ENVBs for the files that require closing. IBMBOCA then closes these files. This may involve completing I/O operations, and hence calling the transmitter. After handling any necessary transmission, IBMBOCA disassociates the file from the data set.

The ENVB is required if the LEAVE or REREAD option is in effect.

For implicit closing, the chain of open files starting in the TCA is scanned to determine which files must be closed. The addresses of the FCBs of these files are then passed to the close routine.

For an explicit close, it is necessary to set the address in the pseudo-register vector to point, once more, to the dummy FCB. This allows implicit opening to be handled should the file be opened again. (See "IMPLICIT OPEN" later in this chapter.)

When IBMBOCA has finished, it returns control (via IBMBOCL) either to compiled code (for an explicit close statement) or to the termination routine (for the end of the program). The code and control blocks generated for a CLOSE statement are summarized in figure 8.10.

Implicit Open for Library -Call I/O

Compiler Output

There is no compiler output for an implicit open, because it is not always possible to predict which transmission statements will cause implicit opening of a file.

Execution

Implicit opening is handled by manipulation of addresses (see figure 8.11).

When IBMBRIO is called for a transmission statement, it executes a test-under-mask (TM) instruction against a set of flags held at an offset from the address held in the pseudo-register vector. The address held in the pseudo-register vector depends on whether the file is open. If the file is open, the pseudo-register offset contains the address of the FCB for the file. If the file is not open, the pseudo-register offset contains the address of a dummy FCB in the program management area.

The address is set during program initialization to point to the dummy FCB, and is reset to the dummy FCB whenever a file is closed.

The first word in the dummy FCB is a set of statement validity flags. These are all set to zero. Consequently any TM instruction executed by IBMBRIO will give a negative result. The second word of the dummy FCB is the address of an entry point in the open/close bootstrap module. If the TM instruction yields a negative result, IBMBRIO branches to the address held immediately after the statement validity flags. Consequently when an attempt is made to execute a transmission statement on a file that is not open, control passes automatically to the open routines.

The open routines open the file, and set up an FCB and DCB for the file. The address of the FCB is placed in the pseudo-register offset, and execution of the statement is reattempted by branching once more to IBMBRIO.

Error Conditions in Transmission statements

To provide PL/I error handling facilities with the minimum possible overhead to error-free programs, transient-library modules are used. These are not loaded unless an error occurs. Two modules are available for every file type except VSAM:

1. The ENDFILE routine, IBMBREF, which can deal only with the ENDFILE condition.

2. A general error module capable of handling all conditions that may arise, including ENDFILE, but loaded only if the TRANSMIT, RECORD, KEY, or ERROR condition occurs. (See figure 8.12.)

Record I/O error module	File types
IBMBREA	Consecutive buffered
IBMBREB	Indexed
IBMBREC	Regional, consecutive unbuffered, and transient
IBMBREE	VSAM
Endfile module	
IBMBREF	All SEQUENTIAL/INPUT/ UPDATE file types (excluding VSAM)

Figure 8.12. Record I/O error modules

This method is used because the short ENDFILE module gives faster execution to those programs that use the ENDFILE condition to handle program flow. The transient error modules for all file types are identified by the six letters IBMBRE followed by a further single character (see figure 8.13).

If a transmission error occurs, the transmission error routine within the transmitter will be entered, whether an in-line or library-call statement is being executed. The transmission error routine has been nominated in the SYNAD exit address placed in the DCB by the OPEN routines. Similarly, if end-of-file occurs, the end-of-file routine within the transmitter will be executed. Record and key errors are detected either by the transmitter or by compiled code.

When any of the errors or PL/I conditions mentioned above occurs during the execution of a record I/O statement, control is passed to the address held in the word "FERM" in the FCB. The address may be any one of the following:

- The address of IBMBREF, the ENDFILE module.
- The address of the general error module for the file type.
- The address of a bootstrap routine, IBMBRIOB. This routine constructs the name of an error module by taking the skeleton IBMBRE*A and replacing the "*" by the letter in the single character field "FEFT" in the FCB. IBMBRIO then loads this error module, places the address of the module in FERM, and branches to the module.

So, by changing the contents of the field FEFT, the transmitter can select a particular error module. The contents of FEFT is one of the following:

- A character indicating the name of the general error module for the file type. This character is placed in FEFT during the execution of the OPEN statement.
- The character "F", indicating the name of the ENDFILE module. The contents of FEFT is changed to "F" by the end-of-file routine in the transmitter, which is entered when data management detects end-of-file.

Thus the module loaded by the bootstrap routine IBMBRIOB, and the address placed in FERM, depend on whether end-of-file or another error is the first to occur on the file.

The result of this arrangement is that the general error module can be called in an end-of-file situation. Similarly, the ENDFILE module can be called when another type of error occurs, if ENDFILE was the first condition to occur. To overcome this problem, the general error module contains code to handle ENDFILE, and the ENDFILE module contains code to test for other conditions, and load and call the general error module if appropriate.

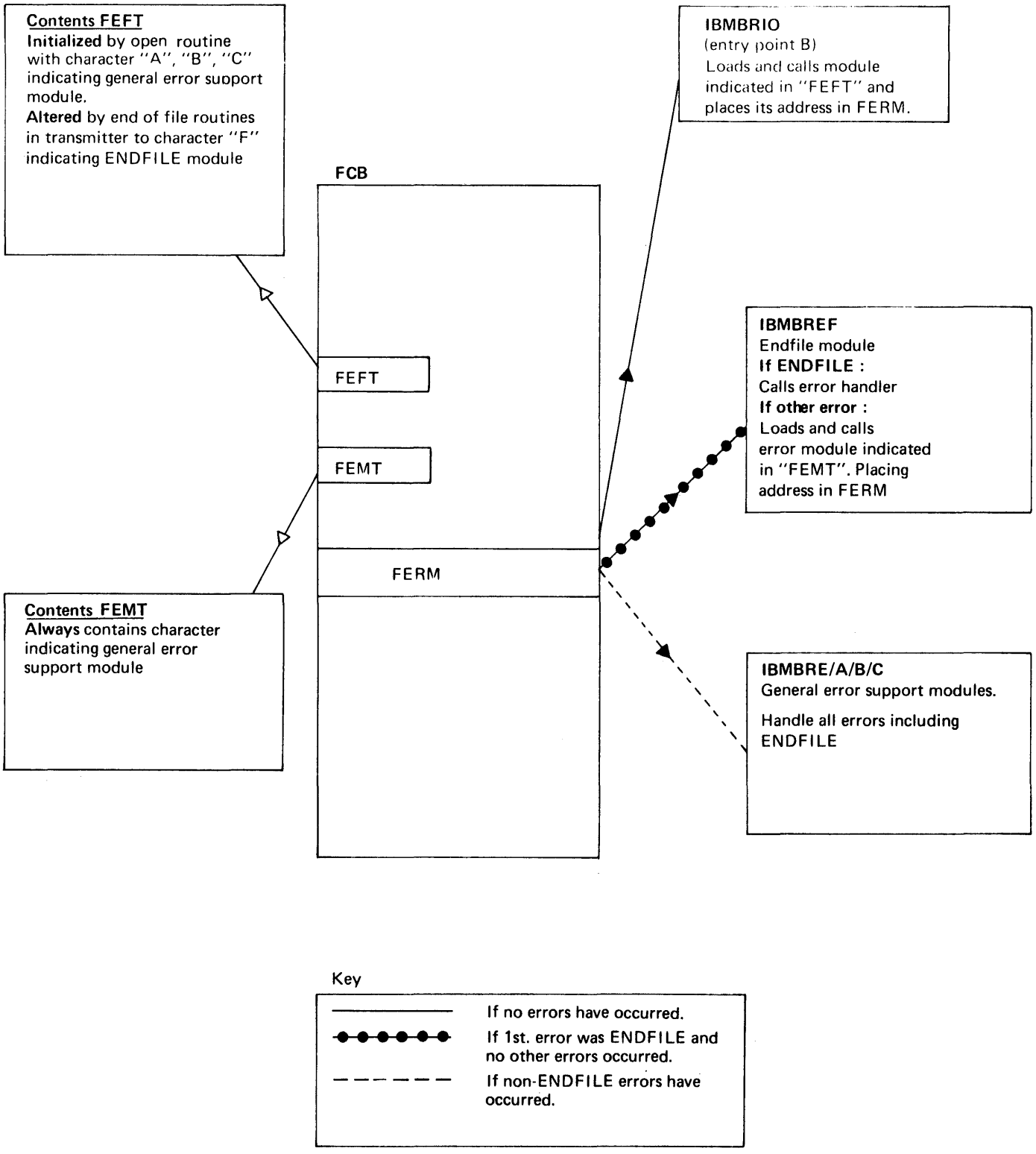


Figure 8.13. The fields used in record I/O error handling

The ENDFILE module constructs the name of the general error module in a similar manner to that used by IBMBRIOB, described above. However, the sixth letter of the name is taken from a field in the FCB called "FEMT." FEMT always holds the character that identifies the general error module for the file. When the name has been constructed, the general module is loaded, its address is placed in FERM, and a branch is made to the module by way of the bootstrap routine in IBMBRIO.

General Error Routines (Transient)

The general error routines set up a parameter list and the relevant built-in function values in the ONCA (described in chapter 7). They then call the resident error handler IBMBERR to handle the condition. If a normal return is made from an on-unit, the general error module will raise any further conditions that have occurred by calling IBMBERR with the appropriate error code. After all conditions have been raised, a return is made to compiled code, or, in event I/O, to the wait module.

ENDFILE Routine

The ENDFILE routine checks to ensure that the situation which has resulted in the call is really end-of-file, and, if so, passes control to the error handler.

TRANSMIT Condition

For certain file types, when a permanent transmission error occurs action must be taken to prevent subsequent issuing of data management macro instructions. To achieve this, addresses are manipulated so that, instead of IBMBRIO calling the transmitter by its primary entry point, it calls an error routine within the transmitter, which in turn calls the error handler to raise the TRANSMIT condition.

In-line I/O Statements

Most transmission statements on buffered consecutive files are implemented by in-line calls to the data management routines (see figure 8.16 for details). Such statements are referred to as "in-line I/O statements." Only READ, WRITE, and LOCATE statements are handled in this way. OPEN and CLOSE statements are always executed by library calls.

Control Blocks

For in-line I/O statements, the only control blocks that are set up are the FCB and DCB. The request control block, record descriptor, and key descriptor are not required as they are merely parameters for library subroutines.

Executable Instructions

For in-line I/O, a call is made direct to the data management routine whose address is held in the FCB. In addition to calling the data management routine, compiled code moves the data as necessary to or from the record variable, or sets appropriate pointers. Compiled code may also check for the RECORD condition.

For U-format and V-format records, compiled code does not call data management direct. Instead a call is made to small routines within the PL/I transmitters. These routines are addressed through the field in the FCB that normally addresses the data management routines. This field is initialized by the open routines when U-format or V-format records are used on the file. The compiler can thus produce the same code for all record types.

For certain types of blocked file, deblocking is handled by compiled code. Fields in the DCB hold the address of the current record, the address of the end of the block, and the record length. Before a call is made to data management, a check is made to see whether the end of the block has been reached. This is done by adding the record length to the current record address. If the resultant address is the end of the block, a call is made to data management for a new block; otherwise, the new address can be taken as the start of the required record.

Error Conditions

If an error occurs during transmission, or if end-of-file is reached, the data management routines will branch to the ENDFILE or SYNAD routines that are held in the PL/I transmitter. (The PL/I

transmitter is always loaded by the open routines.) The ENDFILE and SYNAD routines set an error flag in the FCB, and return to compiled code, normally via the data management routine. If the error flag is on, or if the RECORD condition has occurred, compiled code branches to IBMBRIOD. This results in a call being made to the transient error module.

Typical code produced for an in-line I/O statement is shown in figure 8.14.

Implicit Open for In-Line Calls

Implicit opening for in-line calls is handled in a similar way to that used for library calls.

The field that, in a normal FCB, points to the data management transmitter, in the dummy FCB points to the open/close bootstrap routine, IBMBOCL (see figure 8.11). This results in a branch being made to the OPEN routines when an attempt is made to access a file that is not open. When the open routines have been executed, the address in the pseudo-register vector is altered to point to the FCB that has been created for the file.

If the file is successfully opened, a test is made to see whether the entry to IBMBOCL was for an in-line call and, if it was, control is passed to the data management address held in the DCB. This causes the data management module to be entered and a return made to compiled code.

A further problem arises over deblocking, for certain blocked files, before data management is called, a test is made to see whether the end of the block has been reached. For such files, values are placed in the dummy FCB that ensure that if the test for end-of-block is made before the file has been opened, the test will reveal an apparent end-of-block. A branch will therefore be made to the transmitter field in the dummy FCB, and control will pass to the open/close bootstrap routine.

```
                          SOURCE

1          EXAMPLE:PROC OPTIONS (MAIN);
2    1                 DCL LINE FILE RECORD INPUT
                           ENV(FB,RECSIZE(80),BLKSIZE(400),TOTAL),
                           CARD CHAR (80);
3    1                 READ FILE (LINE) INTO (CARD);
4    1                  END;
```

```
* STATEMENT NUMBER  3
00005E  18 72                      LR    7,2            Save program base
000060  58 90 3 01C                L     9,28(0,3)      Load R9 address of DCLCB
000064  18 B9                      LR    11,9           Load R11 DCLCB
000066  58 10 C 004                L     1,4(0,12)      Load R1 PRV
00006A  5A 10 B 000                A     1,0(0,11)      Add PRV offset in DCLCB to address in R1
00006E  58 20 1 000                L     2,0(0,1)       Point R2 at FCB
000072  58 10 2 014                L     1,20(0,2)      Point R1 at DCB
000076  18 81                      LR    8,1            Save address of DCB
000078  58 10 8 04C                L     1,76(0,8)      Pick up pointer to current record (start of deblocking)
00007C  4A 10 8 052                AH    1,82(0,8)      Add logical record length to access required record
000080  59 10 8 048                C     1,72(0,8)      Compare with end of buffer
000084  47 40 7 03A                BL    CL.2           Branch around data management call if new buffer not required
000088  18 18                      LR    1,8            Restore DCB address if new buffer required
00008A  41 80 3 020                LA    8,32(0,3)      Pass abnormal return address (CL.3) in R8 for error handling
00008E  58 F0 2 01C                L     15,28(0,2)     Load address of data management routine
000092  05 EF                      BALR  14,15          Branch and link to data management routine
000094  47 F0 7 03E                B     CL.4           Branch around next instruction
000098                    CL.2     EQU   *              Label branched to if no data management call
000098  50 10 8 04C                ST    1,76(0,8)      Point R1 at required record
00009C                    CL.4     EQU   *
00009C  D2 4F D 0A8 1 000          MVC   CARD(80),0(1)  Move record into record variable
0000A2                    CL.3     EQU   *
0000A2  91 80 2 02C                TM    44(2),X'80'    Test for errors
0000A6  47 E0 7 052                BNO   CL.5           branch if no errors
0000AA  58 F0 3 014                L     15,A..IBMBRICD if errors call error bootstrap routine
0000AE  05 EF                      BALR  14,15
0000B0                    CL.5     EQU   *
0000B0  18 27                      LR    2,7            restore program base
```

Figure 8.14. In-line I/O transmission statement

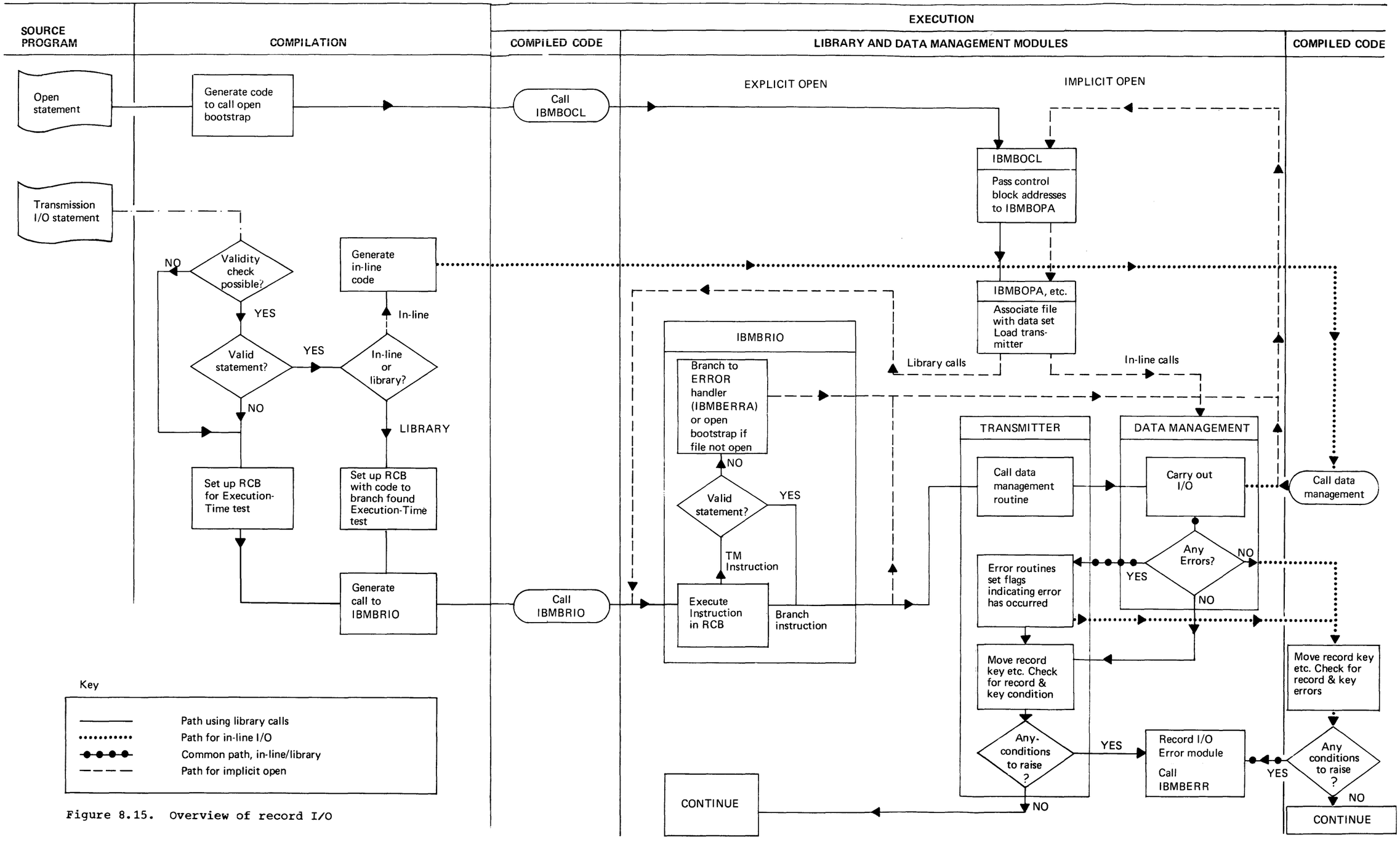
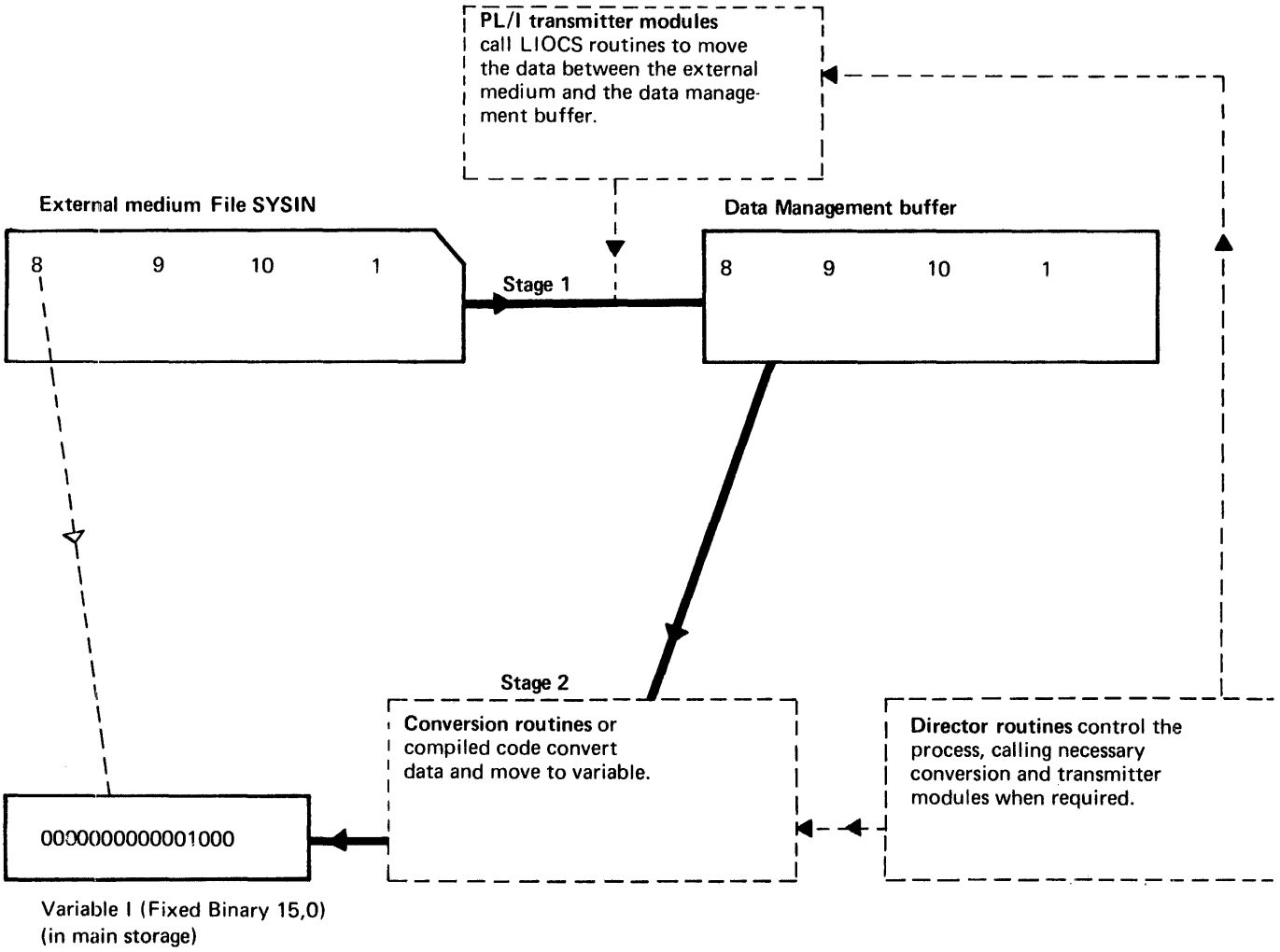


Figure 8.15. Overview of record I/O

File type: Consecutive buffered (TOTAL option used)		
Record type: F,FB		
Statement	Record variable requirements	ENVIRONMENT option requirements
READ SET	None	None
READ INTO	Length known at compile time (max. length if a varying string or area*)	RECSIZE known at compile time SCALARVARYING option if varying string
WRITE FROM (fixed string)	Length known at compile time	RECSIZE known at compile time
WRITE FROM (varying string)		RECSIZE known at compile time SCALARVARYING option used
WRITE FROM Area*		RECSIZE known at compile time
LOCATE A	Length known at compile time (max. length if varying string or area*)	RECSIZE known at compile time SCALARVARYING if varying string
Record type: U,V,VB		
READ SET	None	Not BACKWARDS
READ INTO	Length known at compile time (max. length if varying string or area*)	RECSIZE known at compile time SCALARVARYING if varying string
WRITE FROM (fixed string)	Length known at compile time	RECSIZE known at compile time
WRITE FROM (varying string)		RECSIZE known at compile time SCALARVARYING option used
WRITE FROM Area*		RECSIZE known at compile time
LOCATE	Length known at compile time (max. length if varying string or area*)	RECSIZE known at compile time SCALARVARYING if varying string
Notes: All statements must be found to be valid during compilation. File parameters or file variables are never handled by in-line code. BLKSIZE may be specified instead of RECSIZE for F, V, and U formats (but not FB, VB).		
* Including structures whose last element is an unsubscripted area.		

Figure 8.16. Conditions under which I/O statements are handled in-line

PL/I Statement: GET LIST(I);



Stream input/output is a two stage process. The data is moved between the external medium and the data management buffer, and between the buffer and the variable. Any necessary conversions are made between the buffer and the variable. The operation is controlled by director modules. The director modules call the appropriate routines to do the transmission and conversion. Transmission is carried out in a similar way to that used for RECORD I/O.

Note that a further input statement will require the value 9 which is already in the data management buffer. *Consequently the transmitter need not be called and a pointer must be kept to the position reached in the buffer.*

Figure 9.1. The principles used in stream I/O

Chapter 9: Stream-oriented Input/Output

Note on Terminology

In this chapter, the terms source and target are used when referring to transfer of data. The source is the point from which the data is taken; the target is the point to which it is moved, possibly in a converted format.

Introduction

PL/I stream-oriented input/output allows the programmer to move data between a PL/I variable and an external medium without any concern about internal and external data types or any attention to record boundaries. Both conversion and record boundary problems are handled automatically.

Although it appears to the programmer that the data is moved directly between the external medium and the PL/I variable, the move is, in fact, a two stage process, as shown in figure 9.1. In the first stage, the data is moved to a data management buffer. In the second stage, it is moved from the buffer to the target. When the data is moved to or from an external medium, a complete record is always moved. When the data is moved to or from a PL/I variable, only as much data as is contained in the variable is moved. The amount of data moved in the one stage need bear no relation to the amount moved in the other. Thus synchronization of the two stages is the principal job in implementing stream I/O.

Transmission between the buffer and the external medium is handled by the routines of OS data management. These routines are called by the PL/I transient library transmitters in the same way as that used in library-call record I/O. The movement between the buffer and the PL/I variables is generally handled by the PL/I conversion routines. The transmitters and the conversion routines are called by director routines. These routines determine which modules are required, and when they are needed.

Data items transmitted by stream I/O are not affected by record boundaries (see figure 9.2). There may be any number of data items in a record, and an item may span any number of records. Because the data management routines make only one record available to the program at any one time, a method is needed to build up complete items if they span the record boundary. Similarly, because GET and PUT statements may read or write less than a complete record, a method is needed of keeping track of the position reached in the record, so that the next GET or PUT can start from the correct position.

Operations in a Stream I/O Statement

A stream I/O operation can involve any or all of the following operations:

1. Opening the file, and raising the ERROR condition if the statement is invalid.
2. Keeping track of the position in the buffer.
3. Calling the transmitter for a new record.
4. Building in intermediate workspace an item too large to be held in the current record.
5. Determining which conversion is required, and calling the routine to carry out the conversion.
6. Enqueuing and dequeuing on SYSPRINT.

Control of operations (2) through (5) is handled by director routines. For list-directed and data-directed I/O, PL/I library director routines are used. For edit-directed I/O, the job is shared between library routines, compiler-generated subroutines, and compiled code.

Before the director module or director code receives control, an initialization/termination module is called. This module handles item 1 in the list above: checking statement validity, and opening the file if it is not already open. The initialization/termination routine is also called when every PUT statement is completed, to dequeue on SYSPRINT and, for conversational files, to complete the output. The routine is also called on the completion of GET statements with the COPY option, to transmit the data to the copy file.

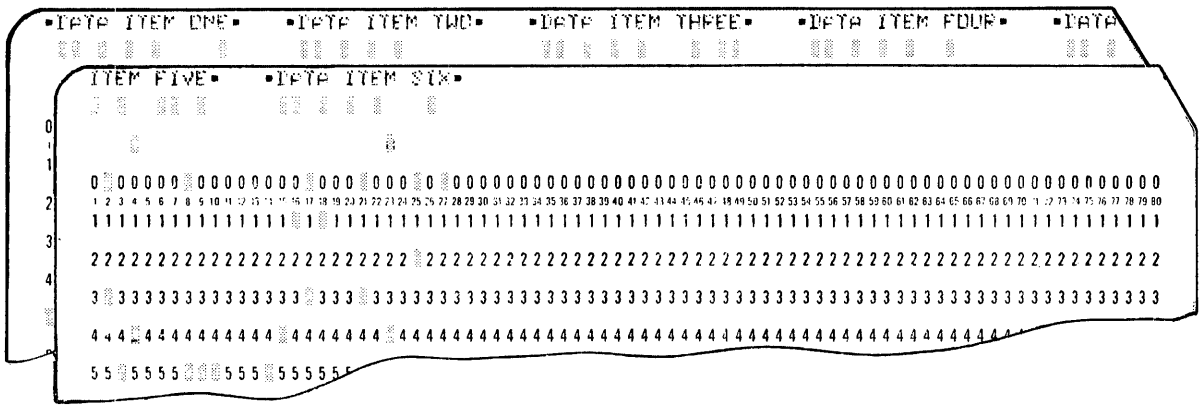


Figure 9.2. Record boundaries do not affect stream I/O

Because there are three modes of stream I/O, the exact situation cannot be defined in a generalized discussion or diagram. However, the basic principles are shown in figure 9.3. The sequence is:

1. A call to the initialization module, to check statement validity, and to open the file if necessary.

2. A return to compiled code, to set up parameters for the director module.

3. A call to the director module to handle any conversion, transmission, and housekeeping problems that may be involved.

4. For PUT statements, a terminating call to the initialization/termination routine to dequeue on SYSPRINT.

Stream I/O Control Block (SIOCB)

To simplify communication between the large number of routines that may be used in a stream I/O operation, a control block is set up for the duration of the execution of the stream I/O statement. This control block is known as the stream I/O control block (SIOCB). The contents of the SIOCB are shown in figure 9.4. The SIOCB contains the addresses of the source and target (or their locators), and of the DEDs of the source and the target. The SIOCB is passed directly to the conversion routines. The first four words contain the parameters expected by the conversion routines.

File Handling

In stream I/O, file organization is always sequential and the access method used is the queued sequential access method (QSAM).

Transmission

Transmitters are called by the director modules or, in certain cases, by the initialization module, or by the close module to complete transmission when the program is terminated.

As with record I/O, transmitters call

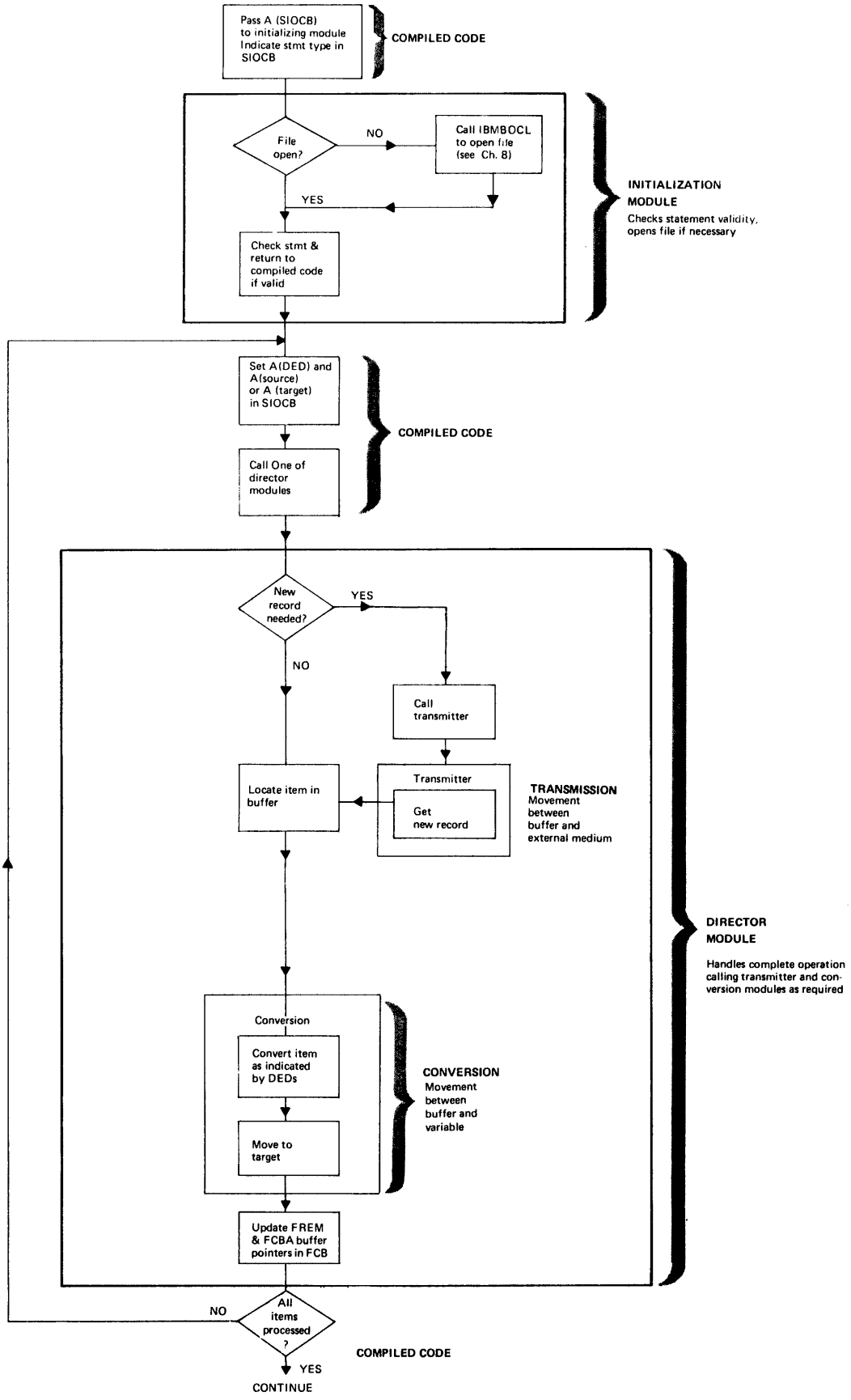


Figure 9.3. Simplified flow diagram of a stream I/O statement

data management modules. The PL/I transmitters contain the EODAD and SYNAD routines, which are entered when end-of-file or other errors are detected in the routines. Nine different transmitter modules are used in stream I/O; these include two for conversational files. The stream I/O transmitters are listed in the summary of subroutines at the end of this chapter.

SSRC	Address of source or source locator
SSDD	Address of source DED
STRG	Address of target or target locator
STDD	Address of target DED
SFLG	Flag bytes
SFCB	Address of FCB for file
SRTN	Abnormal return address (next statement)
SAVE	Save word used by compiler
SCNT	Count of items transmitted (Halfword)
SOCA	Address of ONCA
SSTR	Area present only for GET or PUT STRING, to hold a dummy file control block. (27 fullwords)

Figure 9.4. Stream I/O control block (SIOCB)

Opening the File

The same basic method is used for opening the file as is used for record I/O. During compilation, a declare control block (DCLCB) and an environment control block (ENVB) are set up. An open control block (OCB) is also set up if any environment options are declared in the OPEN statement. At open time, the information addressed from the DCLCB, ENVB, and the OCB (if any) is merged with any information in the DD statement, and an FCB is set up. The PL/I transmitter is loaded, and its address placed in the FCB. A DCB, addressed from the FCB, is set up. The DCB contains the address of the data management transmitter. Finally, the address of the FCB is placed in the pseudo-register vector.

Implicit Open

Implicit opening is handled by the initialization routines, which check to see whether the file is open and, if not, call the open/close bootstrap routine IBMBOCL.

The FCB for stream I/O is similar to that used for record I/O. However, it contains certain additional fields which are needed only for stream I/O. The most important of these fields are the buffer control fields. The format of a stream I/O FCB is shown in appendix A.

Keeping Track of Buffer Position

Two fields in the FCB are used to keep track of the position which has been reached in the data management buffer, and to indicate when a new record will be required. These fields are the buffer control fields:

1. FCBA - points at the position reached in current record.
2. FREM - number of unused bytes remaining in the record.

FCBA points at the position reached in the record and enables the director routines to identify where the next input item must be read from, or where the next output item must be written. FREM contains the number of bytes left in a record. It enables the director modules to determine when a new record will be required, and whether an item is too large to be held in the remainder of the record and will consequently require intermediate workspace. Figure 9.5 illustrates the use of FCBA and FREM.

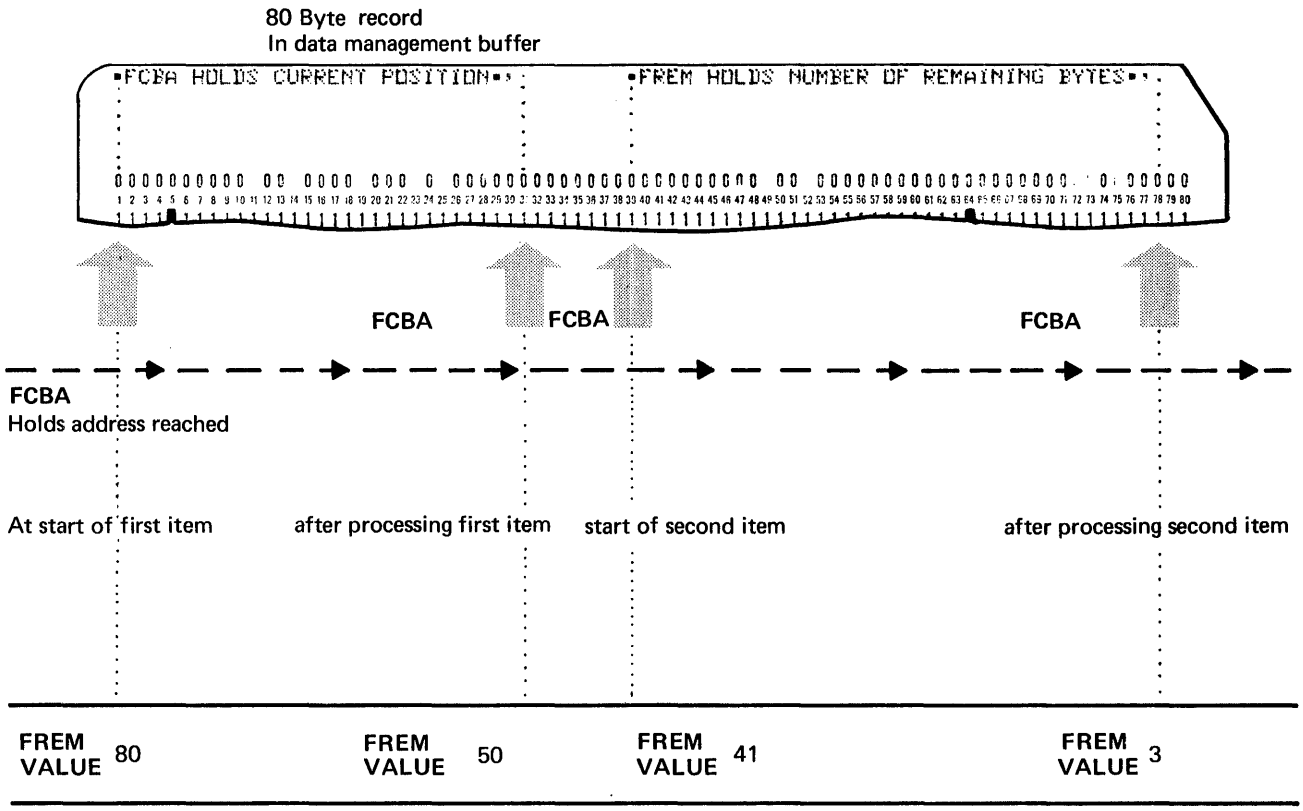
Enqueuing and Dequeuing on SYSPRINT

Because SYSPRINT is used as the standard file for error messages, it is necessary for output to SYSPRINT to be enqueued. This prevents error messages from one task in a PL/I program interrupting other output to SYSPRINT from another task.

When SYSPRINT is used it is enqueued by the initialization routine. When any PUT statement is completed, regardless of the output file, a call is made to the

PL/I STATEMENT:

```
GET FILE (SYSIN) LIST (A, B);
```



FREM holds number of remaining bytes

Figure 9.5 The use of FREM and FCBA in recording buffer position

PUT LIST (A)

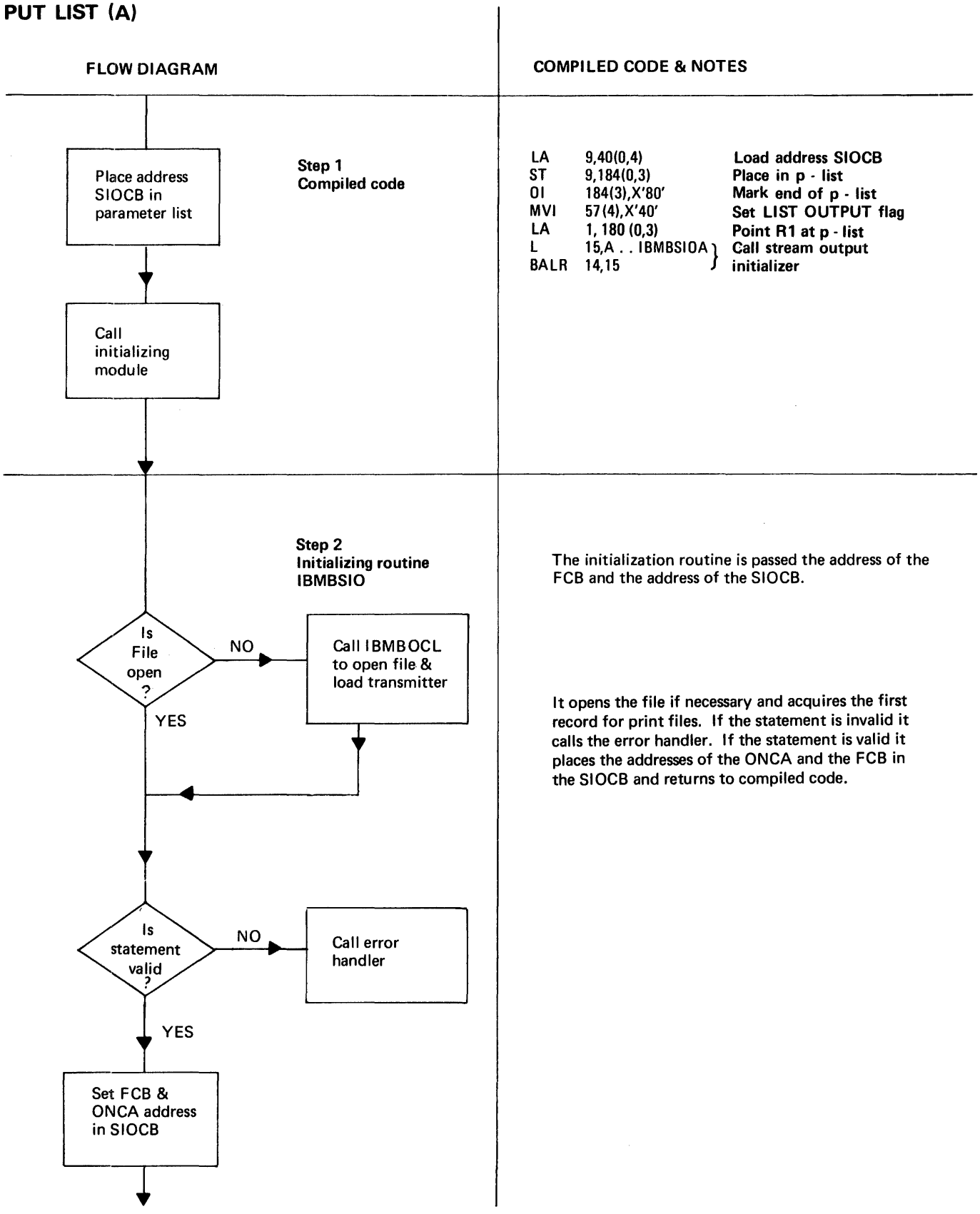


Figure 9.6. (Part 1 of 2). Flow of control through a PUT LIST statement

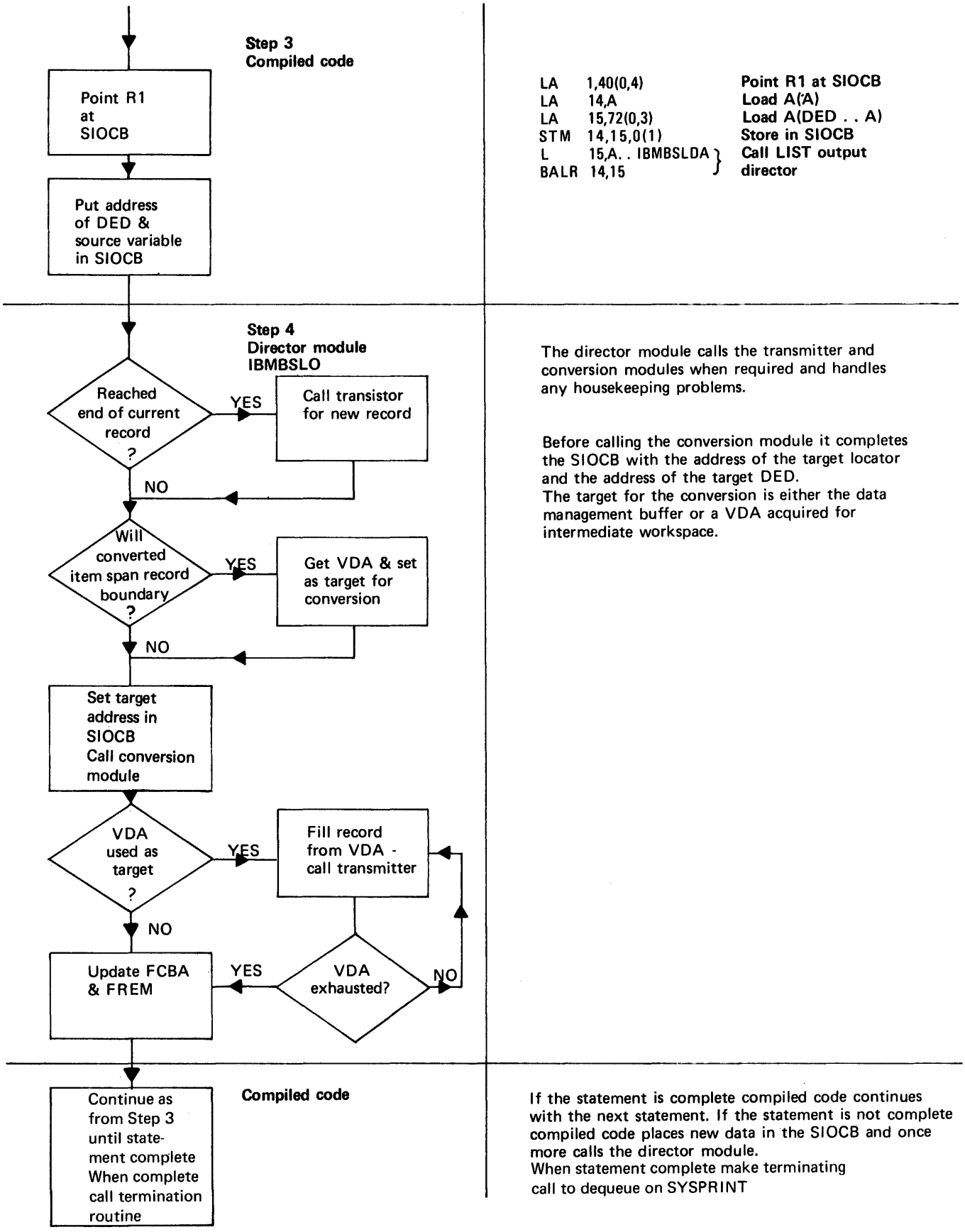


Figure 9.6. (Part 2 of 2). Flow of control through a PUT LIST statement

initialization/termination routine. This routine then checks to see if SYSPRINT has been enqueued and, if it has, dequeues it, by calling the DEQ routine.

Handling the Conversions

Conversions in stream I/O are normally handled by the library conversion package. The conversion package, described in chapter 10, consists of conversion routines and conversion director routines. Conversion director routines examine the DEDs of the source and the target passed in the argument list, and determine which entry point of which conversion module is required. Each possible conversion has a unique entry point in one of the conversion routines. For stream I/O, the argument list passed is contained in the first four words of the SIOCB.

A number of conversion director modules are used exclusively by edit-directed stream I/O. These are called external conversion directors, and are listed in the summary of subroutines at the end of this chapter. Each module corresponds to a particular format of input or output. When the type of input or output has been determined by the director modules, the appropriate conversion director routine can be called to handle the conversion.

In edit-directed I/O, the conversion required is normally predictable during compilation, because it is implied in the format list. Consequently, the conversion modules can be called from compiled code rather than from the stream I/O director routines. A third possibility is that compiled code will handle the conversion in-line.

When a library conversion module is required by compiled code, the conversion director module may be called, or the conversion module itself may be called directly. When the conversion module is called, compiled code must carry out the jobs normally handled by conversion director modules, that is, setting up a number of fields that are used in handling the CONVERSION condition and other PL/I exceptional conditions.

Handling GET and PUT Statements

There are considerable differences in detail between the handling of GET and PUT statements for the three different modes of stream I/O. A generalized impression is given in figure 9.3 and summarized above.

This chapter first covers the implementation of list-directed GET and PUT statements in some detail, and then highlights the differences for data-directed and edit-directed I/O.

List-directed GET and PUT Statements

PUT LIST Statement

Implementation of a list-directed output statement is shown in figure 9.6. The process consists of five steps:

1. Compiled code calls the initialization routine, passing the address of the DCLCB and of the SIOCB. Flags indicating the statement type have been set in the SIOCB by compiled code.

2. The initialization routine, IBMBSIO, calls the open routine if the file is not open, and checks the validity of the statement. If the statement is invalid, a branch is made to the error handler, passing an error code indicating "invalid statement." This results in a message being generated, and the ERROR condition being raised. If the statement is valid, control is returned to compiled code.

   IBMBSIO also handles any format options, by calling the formatting module IBMBSPL. Control then returns to compiled code.

3. Compiled code places the address of the source (or its locator, if the item is a string) and the address of the source DED in the SIOCB. (See chapter 4 for information on locators.) Compiled code then calls the director module.

4. The director module completes the SIOCB with the address of the target locator and the address of the DED of the target. The target locator gives the length required for the item. As the target is a character string, a locator will always be used for it. The address of the target is a position in the buffer. For PRINT files, the position is indicated in the tab table, which will either have been set up by the programmer by use of PLITABS, or may be the default tab table in the library module IBMBSTA. For non-print files, each item is followed by a single blank. PLITABS is

PL/I statements

```
DCL A,B;  PUT LIST (A,B);
```

```
* STATEMENT NUMBER  2
00005E  41 90 D 0C8        LA    9,200(0,13)      Pick up address of SIOCB
000062  50 90 3 044        ST    9,68(0,3)        Store in parameter list
000066  96 80 3 044        OI    68(3),X'80'      Mark end of parameter list
00006A  92 40 D 0D9        MVI   217(13),X'40'    Set LIST OUTPUT flag in SIOCB
00006E  41 10 3 040        LA    1,64(0,3)        Point R1 at SIOCB
000072  58 F0 3 024        L     15,A..IBMBSIOA   Branch to initializing module
000076  05 EF              BALR  14,15
000078  41 E0 D 0A8        LA    14,A             Pick up address of A
00007C  41 F0 3 030        LA    15,DED..A        Pick up address of DED..A
000080  41 10 D 0C8        LA    1,200(0,13)      Place address of SIOCB in R1
000084  50 10 D 0C0        ST    1,192(0,13)      Save address SIOCB in temp
000088  90 EF 1 000        STM   14,15,0(1)       Store addresses in SIOCB
00008C  58 F0 3 02C        L     15,A..IBMBSIOA   Call list directed director routine
000090  05 EF              BALR  14,15
000092  41 E0 D 0AC        LA    14,B             Pick up address of B
000096  58 10 D 0C0        L     1,192(0,13)      Point R1 at SIOCB
00009A  50 E0 1 000        ST    14,0(0,1)        Place address B in SIOCB
00009E  58 F0 3 02C        L     15,A..IBMBSIOA   Call list directed director routine
0000A2  05 EF              BALR  14,15
0000A4  58 10 D 0C0        L     1,192(0,13)      Point R1 at SIOCB
0000A8  58 F0 3 028        L     15,A..IBMBSICT   Make terminating call to dequeue on SYSPRINT
0000AC  05 EF              BALR  14,15
```

Note: The DEDs for A and B have been commoned. Consequently the same address is kept in the SIOCB for both calls to the director modules.

Figure 9.7. Code generated for typical list-directed I/O statement

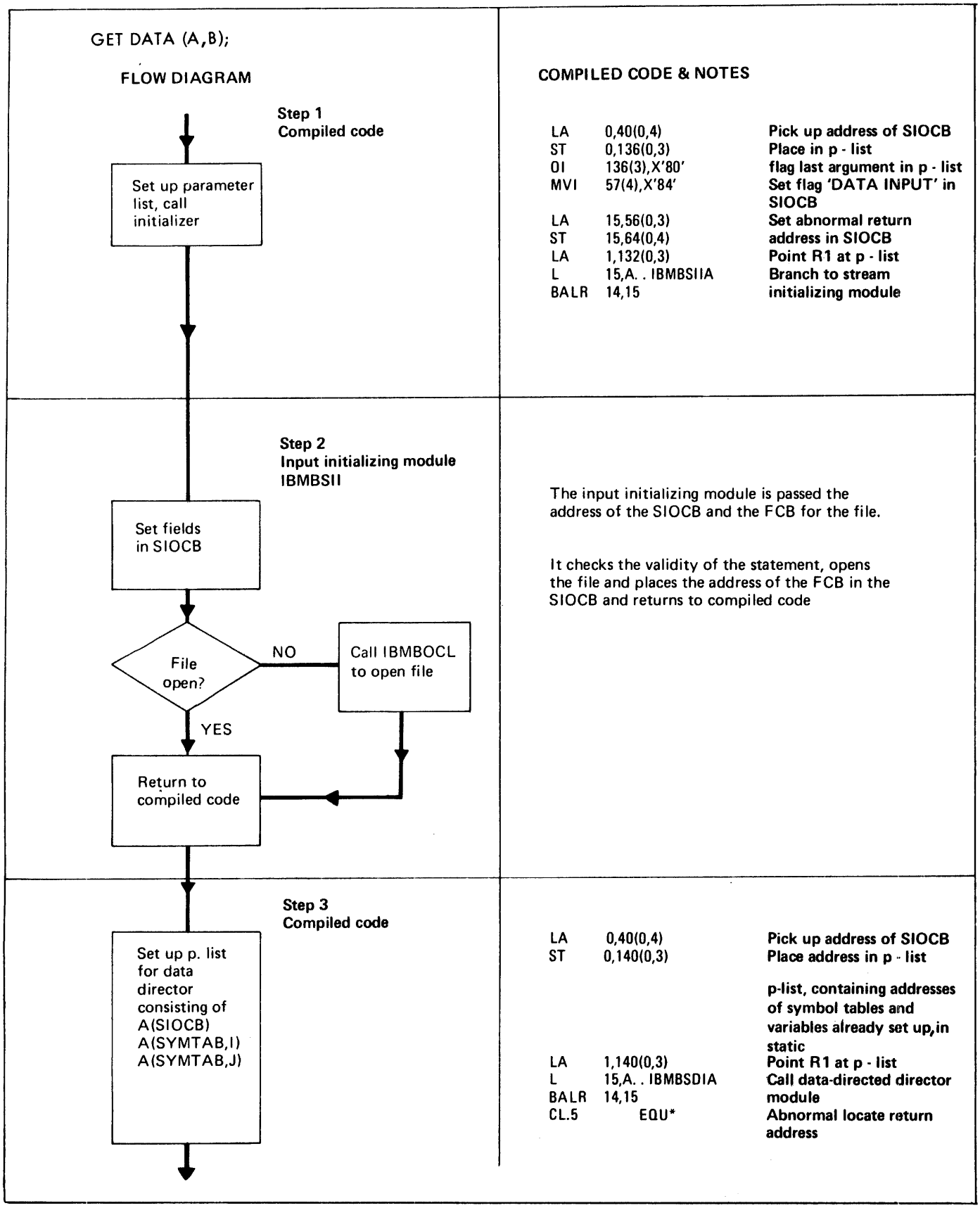


Figure 9.8. (Part 1 of 2). Handling a GET DATA statement

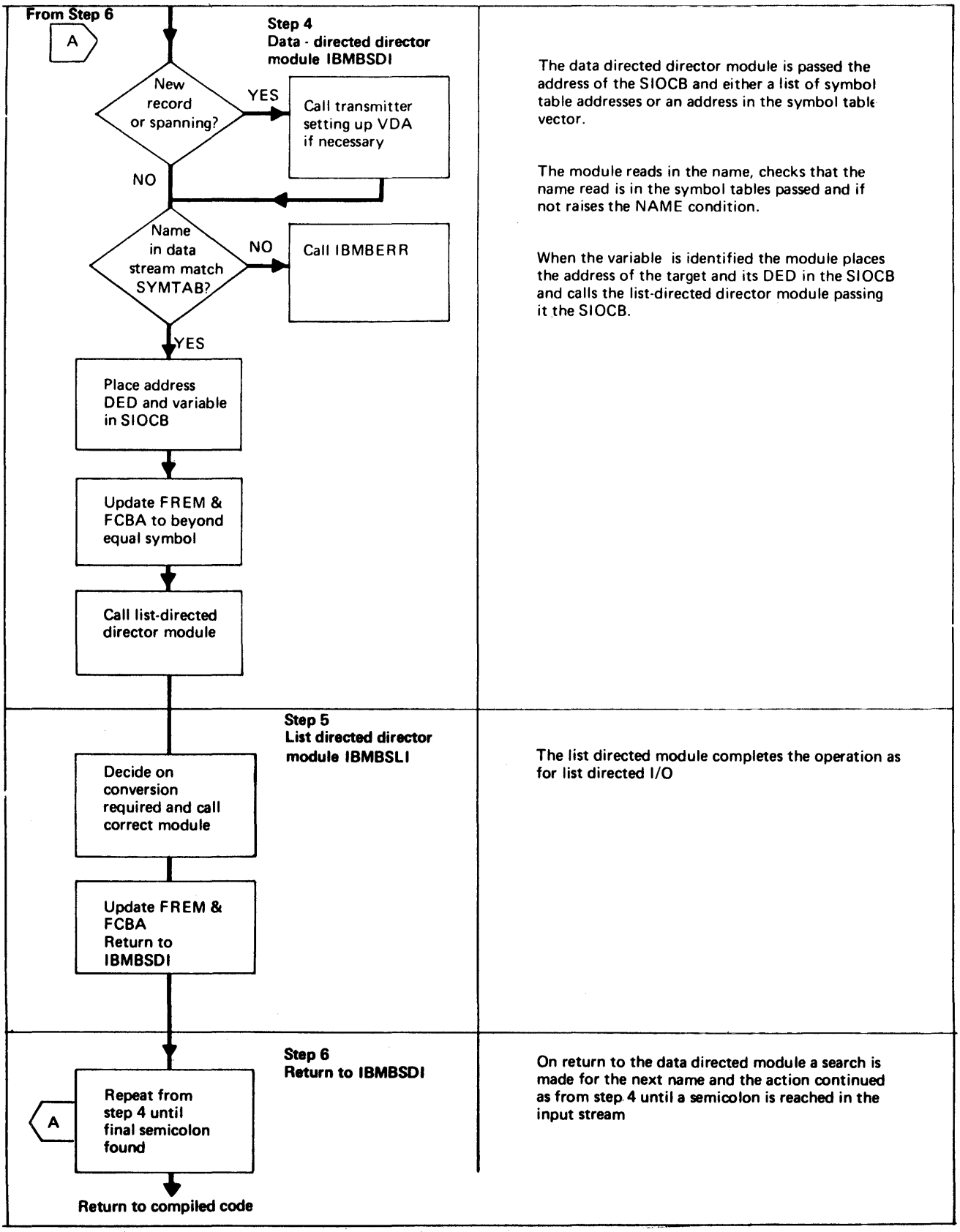


Figure 9.8. (Part 2 of 2). Handling a GET DATA statement

PL/I statements

```
DCL A,B,C;  PUT DATA (A,B,C);
```

RELEVANT SECTION OF STATIC INTERNAL STORAGE MAP

```
000040  00000000              A..DCLCB
000044  80000000              A..TEMP
000048  00000000              A..TEMP
00004C  00000058              A..SYMTAB
000050  0000006C              A..SYMTAB
000054  80000080              A..SYMTAB
000058  8500000100000030      SYMBOL TABLE..A
        000000A800000000
        0001C100
00006C  8500000100000030      SYMBOL TABLE..B
        000000AC00000000
        0001C200
000080  8500000100000030      SYMBOL TABLE..C
        000000B000000000
        0001C300
```

Parameter list for IBMBSIOA

Parameter list for IBMBSDOA

RELEVANT SECTION OF OBJECT PROGRAM LISTING

```
* STATEMENT NUMBER  3
0000AE  41 90 D 0C8          LA     9,200(0,13)       Pick up address of SIOCB
0000B2  50 90 3 044          ST     9,68(0,3)         Store in parameter list
0000B6  96 80 3 044          OI     68(3),X'80'       Mark end of parameter list
0000BA  92 80 D 0D9          MVI    217(13),X'80'     Set data output flags
0000BE  92 01 D 0DA          MVI    218(13),X'01'
0000C2  41 10 3 040          LA     1,64(0,3)         Point R1 at parameter list
0000C6  58 F0 3 024          L      15,A..IBMBSICA    Call initializing routine
0000CA  05 EF                BALR   14,15
0000CC  41 90 D 0C8          LA     9,200(0,13)       Pick up address of SIOCB
0000D0  50 90 3 048          ST     9,72(0,3)         Place in parameter list
0000D4  96 80 D 0DB          OI     219(13),X'80'     Mark end of parameter list
0000D8  41 10 3 048          LA     1,72(0,3)         Point R1 at parameter list
0000DC  58 F0 3 020          L      15,A..IBMBSDCA    Call director routine
0000E0  05 EF                BALR   14,15
0000E2  41 10 D 0C8          LA     1,200(0,13)
0000E6  50 10 D 0C0          ST     1,192(0,13)       Make terminating call to
0000EA  58 F0 3 028          L      15,A..IBMBSIOT    dequeue on SYSPRINT
0000EE  05 EF                BALR   14,15
```

Figure 9.9. Typical data-directed code

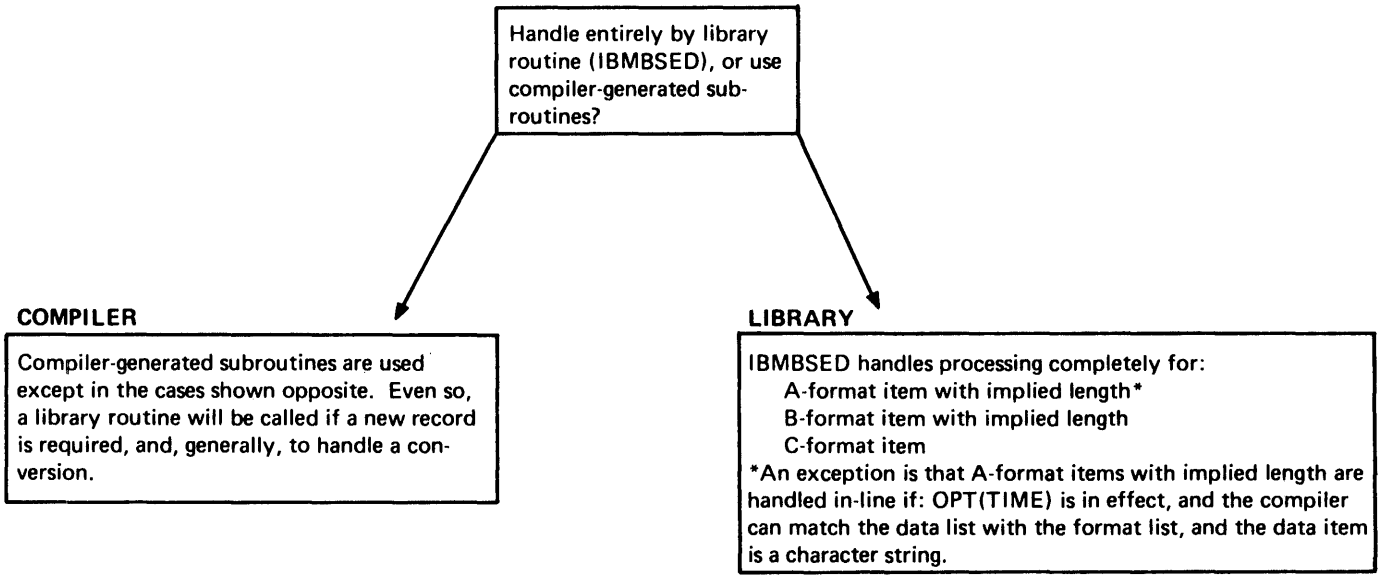


Figure 9.10. The use of the library in edit-directed I/O

PUT EDIT (B)(A);

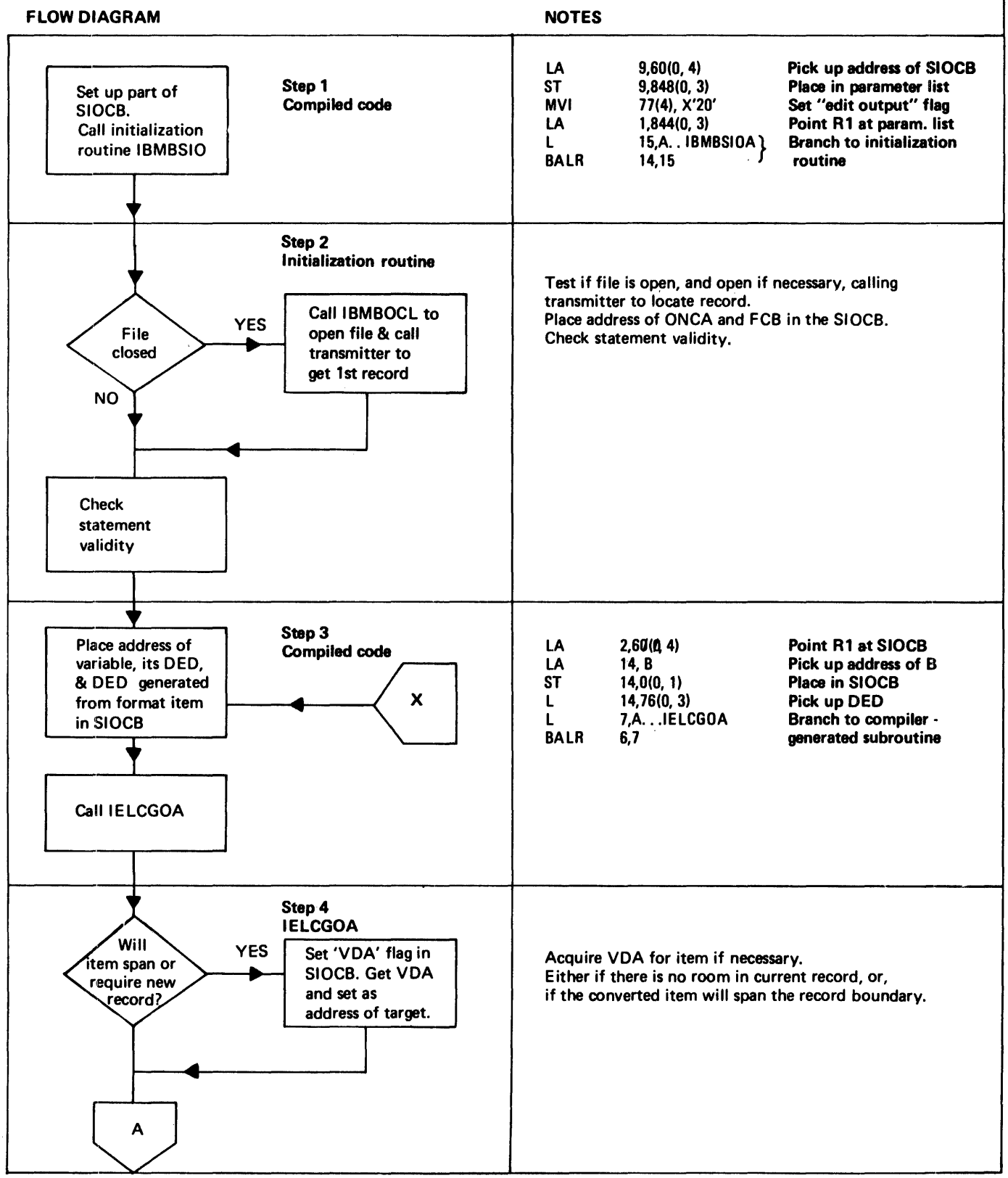


Figure 9.11. (Part 1 of 2). Edit-directed statement with matching data and format lists

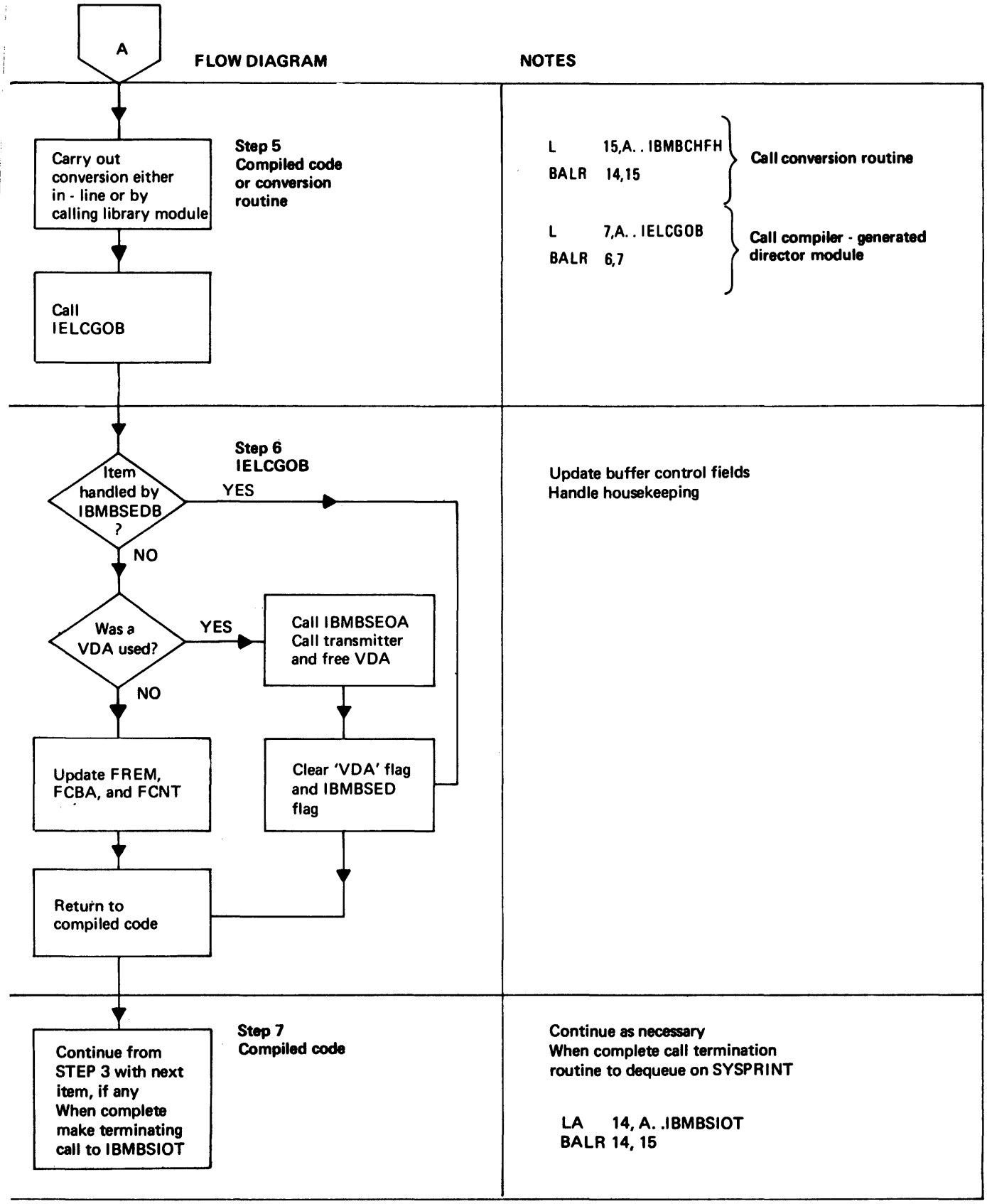


Figure 9.11. (Part 2 of 2). Edit-directed statement with matching data and format lists

addressed from the TCA.

When the starting position for the item has been found, the director module determines whether there is enough space in the output buffer for the converted item. There may not be, for one of two reasons:

a. The end of the buffer has been reached.

b. The converted item will be too large to hold in the buffer.

If the end of the buffer has been reached, the transmitter is called to acquire a new record. If the converted item will be too long to fit in the buffer, intermediate workspace will be needed.

If it is simply a case of acquiring a new record, the director calls the transmitter to acquire it. The director then calls the appropriate conversion routine, passing it the SIOCB as a parameter list. The conversion routine will then move the data from the PL/I variable to the new record in the data management buffer.

If, however, the converted item will span the boundary between the current and subsequent records, intermediate workspace is acquired in the form of a VDA (variable data area - see chapter 6). The converted item is then placed in the VDA. As much of the data as will fit is moved from the VDA into the data management buffer, and a new record is acquired by a call to the output transmitter. The new record is then filled from the VDA. This process is continued until the complete item has been moved into buffers. The buffer pointers FREM and FCBA are updated.

If there are further data items to be handled, a return is made to step (2), and the address of a new source field and its DED are placed in the SIOCB. This process is continued until all items in the data list have been processed.

5. The statement is completed by a call to the initialization/termination routine. This checks to see whether SYSPRINT has been used and, if so, dequeues on SYSPRINT. For conversational files, it also calls the transmitter to transmit any information that is still held in the buffer.

The object code produced for a PUT LIST statement is shown in figure 9.7.

GET LIST Statement

GET LIST statements follow the same sequence, but the transmission is in the opposite direction. The main differences are:

1. If record spanning is involved, the item is assembled in intermediate workspace before it is converted.

2. A locator is built for the source string from the input, and the addresses of the locator and of a character DED for the source are placed in the SIOCB by the director module. The address of the target or its locator and the address of the target DED are placed in the SIOCB by compiled code.

3. Unless the COPY option is being used, no final call is made to the initialization/termination routine.

Data-directed GET and PUT Statements

Data-directed GET and PUT statements follow a similar sequence to list-directed statements, in that there is first a call to the initialization module, followed by a call to a director routine. However, the data-directed director module is passed all the variables involved in the statement rather than one variable at a time, and handles the complete statement without returning to compiled code.

The data-directed director module handles the reading or writing of the names, the equals signs, and the punctuation, and then calls the list-directed director module to handle the value for each variable.

When the data-directed module has identified the location of the variable to or from which the data is to be moved, it calls the list-directed director module which then handles the movement of the value of the variable. When the value of the variable has been transmitted, control returns to the data-directed module, which handles the next name, determines the address of the variable associated with the name, and calls the list-directed director module to handle the transmission of the value. This process continues until the statement is complete. For output, the

director module completes the statement with a final semicolon. Figure 9.8 shows the complete process.

The list-directed director module is called separately for each item. It is passed the SIOCB with the addresses of the source or target (or its locator) and the address of its DED correctly set up by the data-directed director module. The item is then handled as if it were a list-directed item.

Identifying the Name

If a data list is included in the statement, for example:

```
PUT DATA (A,B,C);
```

the source or target variables are identified from a list of symbol tables. If a data list is not included in the statement, for example:

```
PUT DATA;
```

the source or target variables are identified from the symbol table vector.

A symbol table associates a name with the address of a variable. The symbol table vector is a list of the symbol tables known in the external procedure. The items in a symbol table vector are arranged in program block order. When a symbol table vector is used, the address passed is the start of entries for items known in the current block. Symbol tables and the symbol table vector are described further in chapter 4. Their format is shown in appendix A.

The object code produced for a PUT DATA statement is shown in figure 9.9.

Edit directed GET and PUT Statements

Edit-directed I/O differs from the other modes of stream I/O in that the conversions required and the positions in the record where an item is to be placed or will be found are indicated in the format list of the I/O statement.

The format list contains two related types of information:

1. The type and length of the item (e.g., F(3), A(25), etc.), known as data format information.
2. Spacing information (e.g., X(3),COL(70),etc.), known as control format information.

Both types of information are compiled as format DEDs (or FEDs) and are passed by compiled code to the routines that require the information.

Because the information is available during compilation, it is possible for the compiler to determine the conversions that will be required. It is consequently possible for compiled code to call the required conversion or conversion director routine, or to generate in-line conversion code without the assistance of a library director module.

Compiler-generated Subroutines

To further optimize edit-directed I/O, a number of compiler-generated subroutines have been provided. They carry out the following functions:

1. Keeping track of the buffer position, freeing and acquiring intermediate workspace where necessary, and calling the library when a new record is required.
2. Handling X format control items, except where a new record is required.

These compiler-generated subroutines have the advantage over library modules that they are not external, and consequently do not have to follow the external calling conventions.

The compiler-generated subroutines are supported by two types of library director module:

1. Two short modules, IBMBSEO and IBMBSEI, that interface with the transmitter and are called by the compiler-generated subroutines when a new record is required.
2. A routine, IBMBSED, that handles the complete processing of an item (as the director does for list-directed I/O). This routine is called when the item cannot be handled by the compiler-generated subroutines.

The decision on whether to use compiler-generated subroutines or the overall library director module is made at compile time. Figure 9.10 shows the conditions under which each method is used.

A typical edit-directed statement takes

the form:

1. A call to the initialization module to open the file (if necessary), and check statement validity.

2. A call to a compiler-generated subroutine to check whether a new record is required, and, if so, to call the module IBMBSEI or IBMBSEO to acquire a new record by making a call to the transmitter. The SIOCB is completed with source or target DEDs and the addresses of the source and the target or their locators.

3. A call to a conversion module or conversion director, or a compiled-code conversion.

4. A further call to a compiler-generated subroutine, to update the buffer control fields, and free any intermediate workspace if spanning was involved.

5. A terminating call to the initialization/termination routine.

This sequence is illustrated in the annotated flowchart in figure 9.11. Figure 9.12 shows the code generated for a GET EDIT statement.

Handling Control Format Items

Control format items are implemented by calling a formatting module, and passing it the SIOCB containing the address of an FED for a control format item. There are four formatting modules:

1. IBMBSPL: library routine for SKIP, PAGE, and LINE formats and options.

2. IBMBSXC: library routine for X and COLUMN formats.

3. IELCGOC: compiler-generated subroutine for X output items that do not span a record boundary.

4. IELCGIA: compiler-generated subroutine for X input items that do not span a record boundary. (This module also has other functions; see the section "Compiler-generated Director Routines" near the end of this chapter.)

Matching and Non-Matching Data and Format Lists

In the majority of edit-directed statements, the data and format lists can be matched during compilation, since the programmer requires specific conversions for specific variables. However, it is possible to write statements which, because of iteration factors, cannot be matched at compile time.

For example, in the statement

```
PUT EDIT (A,B,C)(N(F(3)),X(10));
```

it is impossible to know at which point the ten-character space indicated by "X(10)" will be required, without knowing the value of N. If the statement had been

```
PUT EDIT (A,B,C)(F(3),X(10));
```

the code would be compiled in the order: handle the conversion of a variable, handle an X item, handle the conversion of a variable, etc., until the data list was exhausted. However, as it is not known at which point the X items will be required in the unmatched statement, it is impossible to compile sequential code to handle the statement. Consequently, the code for each item is compiled separately, and branches are made between the code for data items and the code for format items as the value of the repetition factor indicates. In the example above, the branches would be made when the F item had been executed N times, and when the X item had been executed once.

The code sequences used for matching and non-matching data and format lists are shown in figure 9.13.

Formatting for Print Files

Formatting information such as page size, line size, page length and tab positions for print files are accessed by list- and data-directed director modules from a field TTAB held at offset X'50' in the TCA. The field holds the address of the tab table to be used. That is, either the PLITABS control section, if provided by the user, or the IBMBSTAB control section, if the default is to be used.

The control section PLITABS can be provided by the user either as a control section which is link-edited with the object module or as a PL/I structure declared in his program as PLITABS. This structure is then compiled as a suitable control section by the optimizing compiler.

The programmer may also use the default which is provided as a transient library module loaded by the open routines. The format of PLITABS and its default values are given in the programmer's guide for this compiler.

When the open routines are called, they inspect the TCA to determine whether PLITABS has been provided by the user. If it has not, they load the transient library routine IBMBSTAB, which holds the default tab settings. When the routine is loaded, the address of entry point IBMBSTAB is placed in the TTAB field in the TCA. If PLITABS has been provided by the user, its address will have been placed in TTAB by the linkage editor.

Handling Format Options

Format options (for example, GET SKIP(4), PUT PAGE, GET SKIP LIST) are handled by a call to the appropriate entry point of the initialization routine.

The initializing module calls the formatting module IBMBSPL to carry out the formatting.

PL/I statements

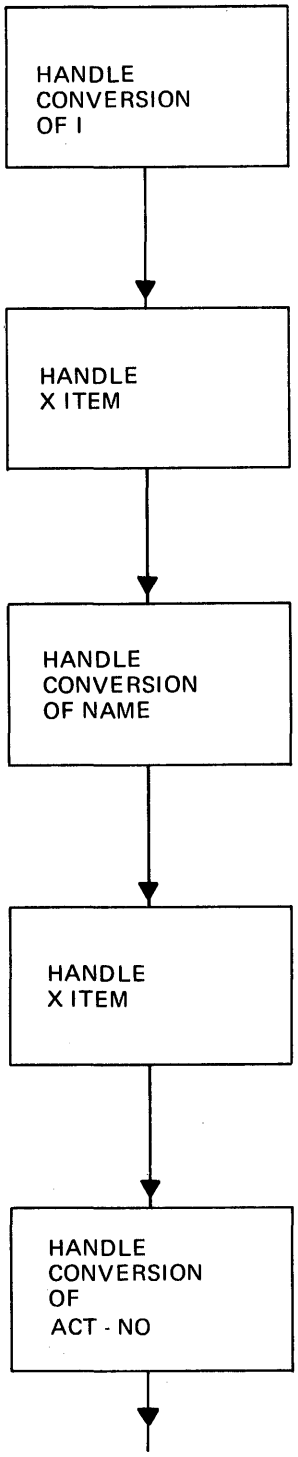
```
DCL A,B;  GET EDIT (A,B) (F(3) ,X(8));
```

```
* STATEMENT NUMBER  3
000062  41 90 D 0C0                LA    9,192(0,13)      Pick up address of SIOCB
000066  50 90 3 054                ST    9,84(0,3)        Store in parameter list
00006A  96 80 3 054                OI    84(3),X'80'      Mark end of parameter list
00006E  92 24 D 0D1                MVI   209(13),X'24'    Set EDIT INPUT flags in SIOCB
000072  41 E0 3 058                LA    14,88(0,3)       Pick up return address (CL.2)
000076  50 E0 D 0D8                ST    14,216(0,13)     Store in SIOCB
00007A  41 10 3 050                LA    1,80(0,3)        Point R1 at parameter list
00007E  58 F0 3 030                L     15,A..IBMBSIIA
000082  05 EF                      BALR  14,15            Call stream I/O initialization routine
000084  41 E0 D 0A8                LA    14,A             Pick up address of A
000088  41 F0 3 038                LA    15,DED..A        Pick up address of DED...A
00008C  41 10 D 0C0                LA    1,192(0,13)      Pick up address of SIOCB
000090  50 10 D 0B8                ST    1,184(0,13)      Save address of SIOCB
000094  90 EF 1 008                STM   14,15,8(1)       Places addresses of A and DED..A in SIOCB
000098  41 E0 3 03C                LA    14,60(0,3)       Point R14 at FED
00009C  58 70 3 00C                L     7,A..IELCGIA     Call compiler generated subroutine
0000A0  05 67                      BALR  6,7
0000A2  58 F0 3 02C                L     15,A..IBMBSFIA   Call conversion director routine
0000A6  05 EF                      BALR  14,15
0000A8  58 70 3 010                L     7,A..IELCGIB     Call compiler generated subroutine
0000AC  05 67                      BALR  6,7
0000AE  41 E0 3 042                LA    14,66(0,3)       Pick up FED of X format item
0000B2  58 10 D 0B8                L     1,184(0,13)      Pick up address of SIOCB
0000B6  58 70 3 00C                L     7,A..IELCGIA     Call compiler generated subroutine
0000BA  05 67                      BALR  6,7
0000BC  41 E0 D 0AC                LA    14,B             Pick up address of B
0000C0  50 E0 1 008                ST    14,8(0,1)        Store in SIOCB
0000C4  41 E0 3 03C                LA    14,60(0,3)       Point R14 at FED
0000C8  58 70 3 00C                L     7,A..IELCGIA     Call compiler generated subroutine
0000CC  05 67                      BALR  6,7
0000CE  58 F0 3 02C                L     15,A..IBMBSFIA   Call conversion director module
0000D2  05 EF                      BALR  14,15
0000D4  58 70 3 010                L     7,A..IELCGIB     Call compiler generated subroutine
0000D8  05 67                      BALR  6,7
0000DA                    CL.2     EQU   *                Abnormal return address
```

Figure 9.12. Code generated for an edit-directed statement with matching data and format lists

MATCHING LISTS

PUT EDIT (I, NAME, ACT. NO)
(F (3),X (3), A (15), X (3), P'ZZZ9');



UNMATCHING LISTS

PUT EDIT (AB, C, D) ((N) F (3), SKIP, A (4));

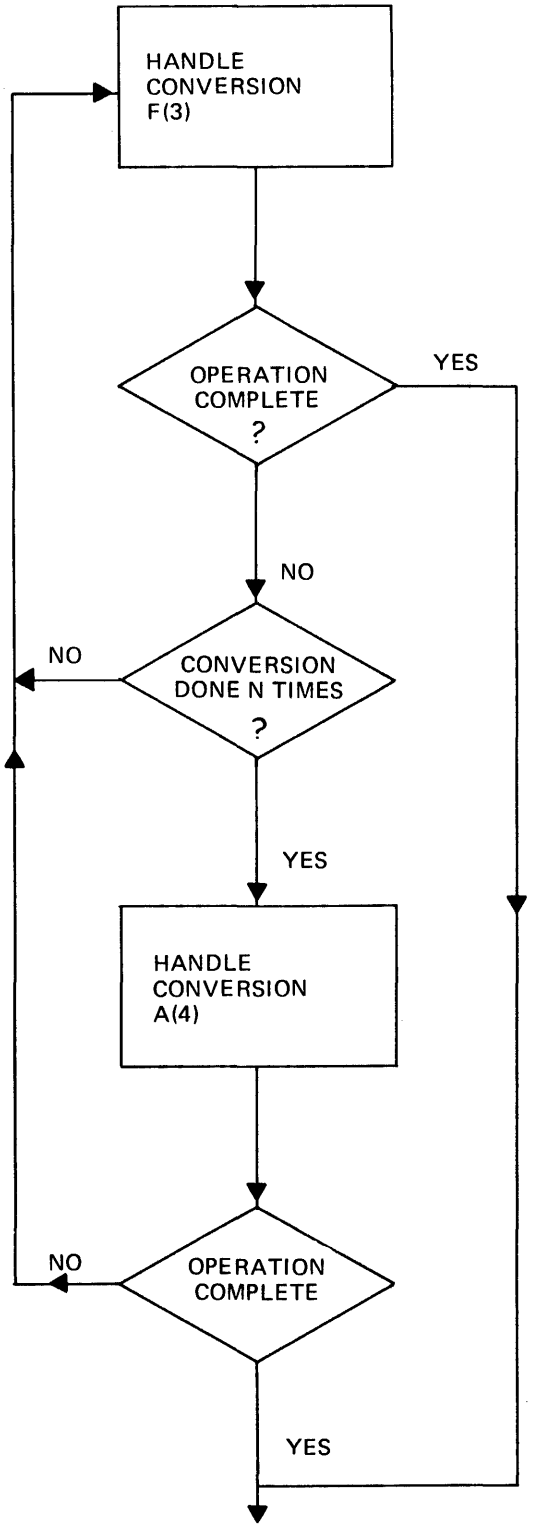


Figure 9.13. Code sequences used for matching and non-matching data and format lists

Input and Output of Complete Arrays

When transmitting complete arrays, it is uneconomical for a return to be made to compiled code after each item has been handled. Accordingly, the list- and data-directed director modules have a facility that enables them to handle complete arrays. The modules access the array multipliers, and handle the indexing from information held in the array descriptors. For edit-directed I/O, each element is handled separately, the necessary indexing being carried out by compiled code.

PL/I Conditions in Stream I/O

The following errors and PL/I conditions are particularly relevant to the implementation of stream I/O: TRANSMIT, CONVERSION, NAME (data-directed input only), ENDFILE, and unexpected end of file. Unexpected end of file occurs when the end of file is reached in the middle of an input item.

TRANSMIT Condition

The rules for raising the TRANSMIT condition in stream I/O are that the condition shall be raised after the assignment or output of the potentially incorrect data item. Thus TRANSMIT can be raised on input for a data item even though the transmitter has not been called during the processing of the statement involved.

When the TRANSMIT condition is detected by the data management routines, control is passed to the error routine in the transmitter, which sets a flag in the FCB indicating a transmission error. For input, the director module inspects this flag, and, if it is set, sets a flag in the SIOCB. TRANSMIT is raised for every item that is taken from a record in the block with which the transmission error was associated. It is raised after each potentially incorrect value has been assigned. For output, TRANSMIT is raised by the transmitter immediately it occurs.

A special entry point, IBMBSEIT, is used by the compiler-generated subroutines to raise the TRANSMIT condition. When called by this entry point, IBMBSEI calls the error handler with the appropriate error code for the TRANSMIT condition.

CONVERSION Condition

The CONVERSION condition is detected by the conversion modules in the PL/I library. (Conversions that could cause the CONVERSION condition are not handled in-line except where "NOCONVERSION" is specified.) CONVERSION is raised by calling a special library module, IBMBSCV. This module analyzes the type of conversion error, and calls the error handler with an appropriate error code. For input, the module also saves the field that caused the conversion; it is necessary to do so, because the field could be lost if an on-unit was entered and a further GET statement was executed on the same file which resulted in a new record being acquired.

NAME Condition

The NAME condition can occur only in data-directed input. It is raised by the data-directed director module when it cannct find a symbol table to match the name read in, or when the name is unobtainable (it might, for example, be out of subscript range.) DATAFIELD is set up, and the file positioned for the next read, before calling the error handler, with the appropriate error code.

ENDFILE Condition and Unexpected End of File

End of file is detected by the transmitter routines, which then enter the SYNAD routine in the transmitter. This routine sets a flag in the FCB. On return to the director modules, the flag is tested and, depending on the situation in which the transmitter was called, ENDFILE or unexpected end of file is raised by calling the error handler.

For unexpected end of file, the ERROR condition is always raised as soon as the end of file is detected. ENDFILE, in the case of list- and data-directed I/O, is not raised until a further attempt is made to read the input file.

Built-in Functions in Stream I/O

The built-in functions that are relevant to stream I/O are COUNT, DATAFIELD, ONCHAR, and ONSOURCE.

ONCHAR and ONSOURCE are dealt with in chapter 10, under the heading "Raising the CONVERSION Condition."

The COUNT built-in function is handled by the director routines. A count of transmitted items for the statement is kept in the SIOCB, and then copied into the FCB after every transmission to or from a PL/I variable.

The DATAFIELD built-in function is handled by the data-directed director routine, which places the address of a string locator/descriptor for the offending field in the ONCA. The field is first moved to a workspace area, as the buffer may get lost if further stream I/O operations take place in an on-unit.

The COPY Option

The COPY option allows input data to be copied onto a specified output file. At the start of a GET statement with the COPY option, a flag is set in the FCB, and the current buffer position is saved in the field FCPM in the FCB.

A resident library routine, IBMBSCP, is used to handle the data, and to transmit it to the copy file by calling the appropriate transmitter. IBMBSCP is called at the end of the GET statement, and during the statement if a new buffer is acquired. The data transmitted to the copy file is that which is held between the pointers FCPM and FCBA. FCBA points to the next byte to be read; FCPM points to the start of the data to be copied. FCPM is updated to point to the start of the new buffer when a transmitter call is made during the execution of the statement. The copy flag is turned off during the terminating call to IBMBSII.

If an interrupt occurs during the execution of a GET statement with the COPY option, it is possible that the terminating call to IBMBSII will be bypassed because of a GOTO from an on-unit, or because the job is terminated. For this reason, a test is made on the copy flag at the start of every GET statement, and when the file is closed. If the copy flag is on, IBMBSCP is called to handle the data. When the data has been transmitted, the flag is turned off.

Handling the Copy File

During the initializing call, IBMBSII determines whether the copy file is open and, if it is not, calls IBMBOCL to open the file. The address of the DCLCB for the copy file is then stored in the FCB of the input file. The data is transmitted to the file by calling the transmitter for the file type.

The STRING Option

The STRING option allows data to be transmitted between a string and one or more PL/I variables by means of a stream I/O statement.

The STRING option is implemented by treating the string specified in the statement as if it were the buffer, and the other PL/I variables as if they were the sources or targets. The differences in housekeeping between string and file operations are resolved by the use of a string housekeeping routine, IBMBSIS. IBMBSIS is called in the place of the stream I/O initialization/termination routine. IBMBSIS sets up a dummy FCB that is initialized so that the correct action is taken should the director modules attempt to read or write beyond the end of the string. After the dummy FCB has been initialized, the director modules are called to convert and move the data as in normal stream I/O.

To implement the string option, compiled code passes the string housekeeping module an extended SIOCB in which the dummy FCB is created. The buffer control fields FCBA and FREM in the dummy FCB are set up as if the string were a record. The field that, in a normal FCB, would hold the address of the transmitter, holds addresses of other sections of code.

For a PUT STRING statement, the transmitter address field is initialized to point to the error handler. Register 1 will have been pointed to the head of the FCB by the caller. The error code for exceeding string size is, therefore, placed at the head of the FCB, and the correct error condition is automatically raised when the branch to the error handler is made.

For a GET STRING statement, the address in the transmitter field is the address of code that sets the end-of-file flag and returns to the caller. This code is held within the dummy FCB.

As far as the caller is concerned, attempting to read beyond the end of the string is equivalent to finding an end-of-file mark in a stream I/O statement. Where the ENDFILE condition or unexpected end of file would be raised for a stream file, a

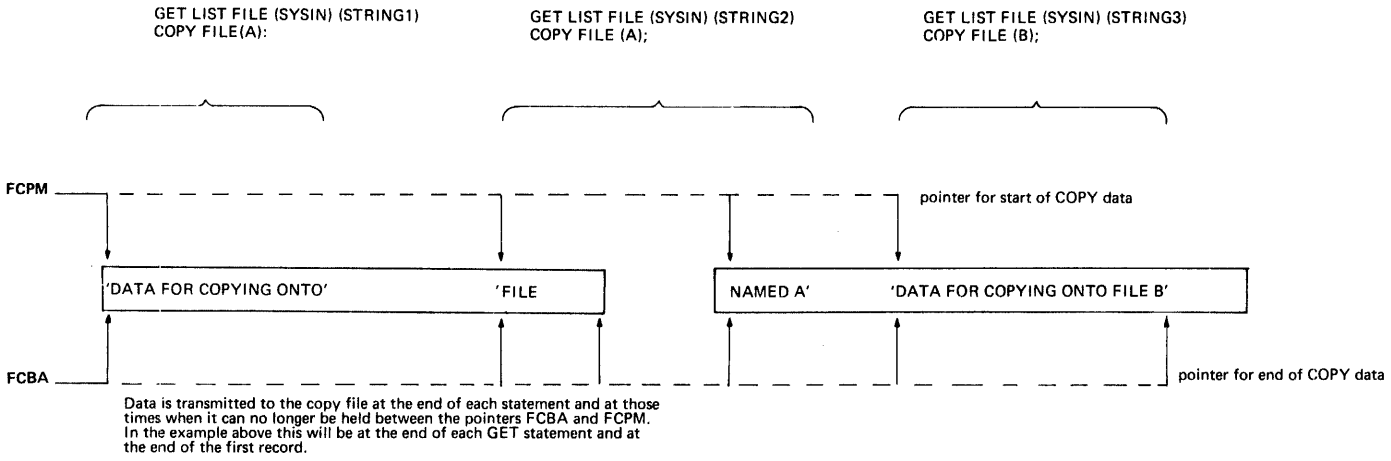


Figure 9.14. The current buffer pointer FCBA and FCPM, the copy pointer, keep track of the data to be copied

'GET STRING SIZE EXCEEDED' message is generated, and the ERROR condition is raised.

Completing String-handling Operations

One or more further calls may be made to the string housekeeping routine IBMBSIS at entry point T, to update the string characteristics after a data item has been transmitted.

PUT statements with fixed-length strings: IBMBSIS is called after the first item has been assigned to the string, to pad the remainder of the string with blanks.

PUT statements with varying strings: IBMBSIS is called to update the length of the string after each data item is transmitted.

GET statements with varying string: IBMBSIS is always called.

The need to make a further call to IBMBSIS is flagged in the SIOCB when IBMBSIS is called to initialize a statement. The library director routines and the compiler-generated subroutines test this flag, and call IBMBSIS if necessary.

The Time-Sharing Option and Conversational Files

When using the time-sharing option, the PL/I programmer can attach the foreground terminal as the input or output device used by one or more stream files.

Three transient library routines are used to implement this facility. Two are transmitters that are used to interface with TSO using the appropriate macro instructions to effect the input and output. The third module is a formatting module that overcomes the special formatting difficulties that arise when working at a terminal.

When the file is opened, the OPEN routine tests every stream I/O file to see whether it is to be associated with a terminal. If the file is to be associated with a terminal, the appropriate conversational transmitter is loaded:

IBMBSIC for input
IBMBSOC for output

A flag is set in the FCB of the file to indicate that the file is a conversational file.

The two transmitter modules handle the input, output, and prompting. Formatting differences between conversational and normal I/O are handled by a transient library routine, IBMBSPC. This routine is called by the formatting routine, IBMBSPL, when a conversational file is being handled.

If a conversational module is used, its address is placed in the TCA loaded-module list.

CONVERSATIONAL TRANSMITTER MODULES

Output Transmitter IBMBSOC

The output module IBMBSOC is similar to other output transmitters except that it interfaces with TSO, and uses the TPUT macro instruction. The macro instruction is used with the WAIT option to ensure proper queueing of output to the terminal.

Input Transmitter IBMBSIC

The input transmitter carries out a similar function to other PL/I input transmitters. However, it also has to handle certain prompting functions, and implements certain facilities required only for conversational output.

Input: Input is achieved by issuing a TGET macro instruction to the TSO control program.

Prompting: Prompting is carried out before every input statement, unless the last character transmitted to the foreground terminal was a colon. At the start of a statement, the prompting sequence is: skip to a new line, print a colon, and skip to the start of the next line. If the GET statement is not completed by the data transmitted from the terminal, a further call to the transmitter will be made by the director module handling the stream I/O. A further prompt is then issued to the programmer. Second and subsequent prompts take the form of a plus character followed by a colon.

Prompts are issued by placing the required prompt-code in a suitable field, and using a TPUT macro instruction with a HOLD option. This ensures that any terminal output from previously executed PUT statements will appear at the terminal

before the user is prompted to enter his input.

The prompt is issued to the foreground terminal irrespective of whether a PL/I output file is associated with the terminal.

FORMATTING

To simplify terminal usage various methods of data input are allowed that do not conform strictly to PL/I language specifications. For example list-directed input need not have a delimiting comma or blank and the trailing blanks need not be entered if a character item in edit-directed I/O does not fill the specified field width.

Formatting Module IBMBSPC

To simplify the use of a terminal, default formatting conventions are assumed. These apply to PAGE, SKIP, and LINE instructions and can be summarized as follows:

> SKIP instructions of 3 lines or less are followed.
>
> PAGE and LINE, and SKIP instructions of more than 3 lines are interpreted as SKIP(3) instructions.

This default formatting can be overridden by the use of a PLITABS structure that specifies a value of 1 or greater for the page length. (PLITABS is described above under the heading "Formatting for Print Files.")

IBMBSPC checks the page-length value in the PLITABS control section. This control section will be either the default taken from the PL/I transient library module IBMBSTAB, or, if the values have been specified by tbe programmer, will be the values in the structure declared with the name PLITABS, or, possibly, a link-edited control section called PLITABS. In the library module IBMBSTAB, the page-length value is zero.

If the page-length value in the PLITABS control section is zero, the formatting conventions described above are used. These are referred to as squashed mode. If the value is greater than zero, normal formatting is undertaken.

The method of formatting used is for IBMBSPC to insert the required number of 'new line' characters in the output buffer, and to call the transmitter to transmit the buffer contents. (In the special case of SKIP(0), backspace characters are used.)

The normal PL/I rules for ENDPAGE apply to formatted terminal output. ENDPAGE is not raised for squashed mode output.

Summary of Subroutines Used

This section gives a summary of the subroutines used in the implementation of stream-oriented input/output. Detailed descriptions of the library modules are given in the relevant program logic manuals.

Ten different types of subroutine are used in stream I/O. They are:

1. Initializing modules
2. Director modules
3. Transmitter modules
4. Formatting modules
5. Conversion modules
6. External conversion director modules
7. Conversational modules
8. The conversion fix-up module (IBMBSCV)
9. The copy module (IBMBSCP)
10. The string housekeeping module (IBMBSIS)

Conversion modules are described in chapter 10 of this manual. The other types of module are dealt with below.

INITIALIZING MODULES

Initializing modules initialize the stream I/O statement. There are two of these modules:

IBMBSII - input initializer
IBMBSIO - output initializer

A further module is used for string handling. See below under "Miscellaneous Routines."

IBMBSII and IBMBSIO are described earlier in this chapter.

DIRECTOR MODULES

Library Director Routines

IBMBSLI - list-directed input
  Entry point A: element item
  Entry point B: complete array

IBMBSLO - list-directed output
  Entry point A: element item
  Entry point B: complete array

IBMBSDI - data-directed input
  Entry point A: with data list
  Entry point B: all known variables

IBMBSDO - data-directed output
  Entry point A: element variables and whole arrays
  Entry point B: single array elements
  Entry point C: all known variables and SIGNAL CHECK output
  Entry point D: CHECK output for a single item
  Entry point T: output a final semicolon only.

Modules Used with Compiler-generated Subroutines

IBMBSEI - edit-directed input
  Entry point A: housekeeping for input item spanning a record boundary.
  Entry point T: raise TRANSMIT for spanning input item

IBMBSEO - edit-directed output housekeeping for output item spanning a record boundary.

Module for Complete Library Control of Edit-directed I/O of a Single Item

IBMBSED
  Entry point A: edit-directed input
  Entry point B: edit-directed output

Compiler-generated Director Routines

For input:

IELCGIA - provides the address of the source of an edit-directed data or X-format item.

IELCGIB - completes the transmission of an edit-directed data item, by freeing a VDA if one was used, updating the COUNT built-in function value, and calling IBMBSEIT if TRANSMIT has been raised.

For output:

IELCGOA - provides the address of the target of an edit-directed data item.

IELCGOB - completes the transmission of an edit-directed data item, updating the buffer items in the DCLCB, counting the data item, and freeing a VDA if one was used.

TRANSMITTER MODULES

The actual movement of the data between the external medium and the buffer area is carried out by a series of seven transmitter modules, which interface with the routines of OS data management. These modules essentially complete the setting up of the DCB, and issue the data management GET and PUT macro instructions, thus reading or writing one record.

One module is used for input, six for output. The output modules are divided into two groups: one group for PL/I print files, the other for all other output files. Both output module groups contain three modules: one for F-format records, one for V-format records, and one for U-format records. All modules interface with the queued sequential access method.

The following transmitters are used:

IBMBSTI - input transmitter

IBMBSOF - output transmitter for F-format records

IBMBSOV - output transmitter for V-format records

IBMBSOU - output transmitter for U-format records

IBMBSTF - print transmitter for F-format records

IBMBSTV - print transmitter for V-format records

IBMBSTU - print transmitter for U-format records

FORMATTING MODULES

Formatting modules control the position of the data on the external medium. There are three formatting modules: two library subroutines, and one compiler-generated subroutine.

Library Subroutines

IBMBSPL - PAGE, LINE, and SKIP format items and options
- Entry point A: PAGE option or format item
- Entry point B: LINE option or format item
- Entry point C: SKIP option or format item

IBMBSXC - X and COLUMN format items
- Entry point A: X format input
- Entry point B: X format output
- Entry point C: COLUMN format input
- Entry point D: COLUMN format output

Compiler-generated Subroutine

IELCGOC - X items, in edit-directed output, that do not span a record boundary.

EXTERNAL CONVERSION DIRECTOR MODULES

The following external conversion director routines are used exclusively in edit-directed I/O:

- IBMBSAI - input A, B, and P character formats
- IBMBSAO - output A, B, and P character formats
- IBMBSCI - input C format
- IBMBSCO - output C format
- IBMBSFI - input F and E formats
- IBMBSFO - output F and E formats
- IBMBSPI - input P format arithmetic
- IBMBSPO - output P format arithmetic

CONVERSATIONAL MODULES

Transmitters:
- IBMBSIC - input transmitter
- IBMBSOC - output transmitter

Formatting module:
- IBMBSPC - formatting module

MISCELLANEOUS MODULES

The other subroutines used in stream I/O are:

IBMBSCV - the conversion fix-up module

IBMBSCP - the copy module

IBMBSIS - the string housekeeping module

Chapter 10: Data Conversion

Note on Terminology

In this chapter, the terms source and target are used when referring to transfer of data. The source is the point from which the data is taken; the target is the point to which it is moved, possibly in a converted format.

The PL/I language specifies situations in which conversion of data types will be carried out. These include the execution of stream I/O and assignment statements, and the evaluation of expressions that include different types of data. The large number of data types allowed in the PL/I language means that some 170 types of conversion are possible. How these conversions are handled by the PL/I Optimizing Compiler depends, to some extent, on the optimization specified for the program.

If optimization has been specified, all conversions that can be handled by in-line code are so handled. If optimization has not been specified, the simpler and more commonly used conversions will be handled in-line, the remainder by the library conversion package.

This chapter describes the library conversion package and explains how in-line conversions are handled. It concludes with a description of how the CONVERSION condition is raised.

Before conversions can be understood, knowledge of the way in which data types are held is necessary. This is summarized in figure 10.1.

Data attributes	Stored internally as
BIT(n)	Aligned: one byte for each group of eight bits or part thereof. Unaligned: as many bits as are required, regardless of byte boundaries.
BIT(n) VARYING	As BIT(n), with two-byte prefix containing current length of string.
CHARACTER(n)	One byte per character.
CHARACTER(n) VARYING	As CHARACTER(n), with two-byte prefix containing current length of string.
FIXED DECIMAL(p,q)	Packed decimal: 1/2-byte per digit, plus 1/2-byte for sign.
FIXED BINARY(p,q)	p <= 15: halfword p>15: fullword
FLOAT DECIMAL(p)	p<=6: short floating-point p>6: long floating-point p>16: extended
FLOAT BINARY(p)	p<=21: short floating-point p>21: long floating-point p>53: extended
PICTURE	One byte for each picture character (except K and V)

Figure 10.1. Internal forms of data types

The Library Conversion Package

The library conversion package consists of some 26 modules and is capable of handling all the conversions that are allowed in the OS PL/I Optimizing Compiler implementation of the PL/I language. All but seven of the modules convert data from one data type to another. As there are approximately 170 possible conversions and only 19 conversion modules, many conversions are done by using a series of modules. For instance, to convert from fixed-decimal to bit-string involves an intermediate conversion to floating-point. The conversion package also contains five control and utility modules, and two modules used for stream I/O. The stream I/O modules move character and bit strings between the data management buffer and the PL/I variable when no conversion is necessary.

A full description of the routines in

the library conversion package is given in the publication OS PL/I Resident Library: Program Logic.

The conversion paths followed for every conversion are known to the compiler, and ESD records are generated for all the modules that will be used. In certain cases, however, the data types involved are not known at compile time. Examples of this are data-directed and list-directed input, and edit-directed input or output when format and data lists cannot be matched. In such cases, the compiler generates ESD records for all conversion modules that could possibly be needed.

Conversion Module Naming Conventions

All names begin with the letters 'IBMB'. The fifth letter is 'C' for conversions, conversion utilities, and the string/arithmetic directors. It is 'S' for the edit-directed format directors. The modules in the arithmetic conversion package have six letter names, the sixth letter being an arbitrary module identifier. The string conversion modules and conversion utilities have seven letter names in which the sixth and seventh letters are mnemonic; The mnemonic codes follow:

X	fixed binary
F	float
I	integer or binary constant if in C module
I	input if in S module
D	fixed decimal
Z	free decimal or float decimal
P	fixed pictured decimal
E	float pictured decimal
H	decimal constant
Y	float decimal constant on output
B	bit
J	bit constant
C	character
Q	pictured character
A	arithmetic
O	output in S module
G	"check" or utility
T	table

SPECIFYING A CONVERSION PATH

When a number of conversion modules need to be used for a certain conversion, it is necessary for there to be some control of the path taken after the first module has been entered. The method used is for each module to have a number of entry points. Each one is entered for a certain type of conversion, and each one implies the subsequent entry points to be invoked for that particular conversion. For instance, the module IBMBCE handles fixed-decimal to fixed-binary conversions. If the module is entered to carry out this conversion, entry point IBMBCEDX is called. However, if it is only an intermediate stage in a conversion from fixed-decimal to bit-string, the entry point IBMBCEDB will be called. When the conversion to floating-point is completed, the conversion to bit will be carried out by the module IBMBCR.

In addition to the use of various entry points to specify the conversion path to be taken, there are two control modules to handle the conversion paths between character-string and arithmetic data.

HOUSEKEEPING WHEN MORE THAN ONE MODULE IS USED

When more than one arithmetic conversion module is used in a conversion, a method of minimizing the housekeeping has been evolved. This avoids saving registers and acquiring workspace for each module entered. The same library workspace is used for all modules in a single conversion operation. The first module in the chain saves the registers and acquires workspace; the last module frees the workspace and restores the registers.

A simple method is used to allow each module to test whether or not it can use the previous module's workspace. A bit at a fixed offset from register 13 is tested. If the module is the first to be called, this bit will be a bit in the calling procedure's DSA, which is always set to zero. If the module is not the first to be called, the bit will be in library workspace and will have been set to one by the previous module if the same workspace can be used. If the module is the first, library workspace will be acquired in the usual manner. If the module is not the first, a branch will be made around this code.

ARGUMENTS PASSED TO THE CONVERSION ROUTINES

Each conversion routine expects a standard set of arguments. These consist of the address of the source and target, and the addresses of the DEDs (data element

Conversion		Comments and Conditions	Optimization	
Source	Target		SIZE disabled	SIZE enabled
Fixed binary	Fixed binary	-	-	-
	Fixed decimal	If either scale factor = 0 and the other factor ≤ 0, the optimization can be 'none'.	time	time
	Floating-point	If source scale factor = 0, the optimization can be 'none' (whether SIZE is enabled or not).	time	time
	Bit string	String must be fixed-length, aligned, and with length ≤2048.	-	not done in-line
	Character string or picture	Source scale factor must be ≤ 0. String must be fixed-length with length ≤256. Picture type 1, 2, or 3.	time	not done in line
Fixed decimal	Fixed binary	If source and target scales have the same sign and are non-zero, the optimization (SIZE disabled) must be 'time'.	-	time
	Fixed decimal	-	-	-
	Floating-point	Source precision must be ≤10.	time	time
	Bit string	Source scale factor must be zero. String must be fixed-length, aligned, and with length ≤2048.	-	not done in-line
	Character string	Source scale factor must be ≥ 0. String must be fixed-length and length ≤256.	time	time
	Picture	Picture type 1, 2, or 3. For picture types 1 and 2 with no sign, optimization can be 'none'.	time	not done in-line
Floating-point	Fixed binary	-	time	not done in-line
	Fixed decimal	Target precision must be ≤9.	time	not done in-line
	Floating-point	Source and target may be single or double length.	-	-
	Bit string	String must be fixed-length, aligned, and with length ≤2048.	time	not done in-line
Bit string	Fixed binary	Source string must be fixed-length, aligned, and with length ≤2048.	-	not done in-line
	Fixed decimal and floating-point	Source must be fixed-length, aligned, and with length <32.	time	not done in-line

Figure 10.2. (Part 1 of 2). Data conversions performed in-line

Conversion		Comments and Conditions	Optimization	
Source	Target		SIZE disabled	SIZE enabled
Picture	Character string	String must be fixed-length with length ≤256.	-	-
	Picture	Pictures must be identical.	-	-
Picture type 1 (See note below)	Fixed binary	Source precision must be <10.	time	not done in-line
	Fixed decimal	If picture has a sign, the optimization must be 'time'.	-	not done in-line
	Floating-point	Source precision must be <10.	time	not done in-line
	Picture	Picture type 1, 2 or 3.	time	not done in-line
Locator	Locator	-	-	-
Label	Label	-	-	-
The word "time" in the columns headed "Optimization" indicates that the conversion is done in-line only if optimization has been specified; "not done in-line" indicates that the conversion is done by library call.				

Figure 10.2. (Part 2 of 2). Data conversions performed in-line

descriptors) for the source and the target. Arguments are passed in a list addressed by register 1. (The source is the variable or constant that requires conversion; the target is the area where the converted result is to be placed.)

The DEDs are used to describe the data type of the element. Those passed to the library conversion package are set up by compiled code in the constants pool. They are described in chapter 4 and fully mapped in appendix A.

COMMUNICATION BETWEEN MODULES

When the conversion path goes through a series of modules, the address of the final target must be retained until the last module is reached.

Temporary targets and DEDs are created for the intermediate results, and these are passed on as the source for the next module. When information is passed between two conversion modules using the same workspace, registers are normally used rather than a parameter list.

In some arithmetic conversions to string, precision data is passed through certain modules that do not themselves need such data.

FREE DECIMAL FORMAT

Because all floating-point data is in binary form, there is no direct representation of the PL/I floating-point decimal format. In order to simplify certain conversions, a simulated floating-point decimal format is employed by the optimizing compiler. This format is termed free decimal (sometimes known as packed intermediate decimal). The format of free decimal is a 17-digit packed decimal mantissa and a fullword binary exponent. Conversions to and from free decimal form

Conversion number	Conversion
2	Fixed-binary to floating-point
3	Floating-point to fixed-binary
4	Fixed-decimal to floating-point
5	Floating-point to fixed-decimal
6	Fixed-binary to fixed-decimal
7	Fixed-decimal to fixed-binary
8	Character-string to fixed-decimal
9	Character-string to floating-point
10	Character-string to fixed-binary
12	Fixed-decimal to character-string
14	Bit-string to character-string
15	Fixed-binary to bit-string
16	Floating-point to bit-string
17	Bit-string to fixed-binary
18	Fixed-decimal to picture type 1
19	Fixed-decimal to picture type 2
20	Fixed-decimal to picture type 3
21	Picture type 1 to fixed-decimal

Note: Conversions numbers 1, 11, and 13 not used.

Figure 10.3. Fundamental in-line conversions

an integral part of the arithmetic conversion package.

In-line Conversions

The optimizing compiler generates in-line code for the more commonly used conversions. Eighteen basic types of conversion are handled in-line. Several of these basic types are used in conjunction, to enable a total of 28 conversions to be handled in-line. The circumstances in which in-line conversions are used are shown in figure 10.2.

An example of the way in which a compiler conversion is used to convert from fixed-binary to fixed-decimal is given below. A list of the eighteen fundamental compiler conversions is given in figure 10.3.

Note about Picture Variables

Not all the picture characters available may be used in a picture involved in an in-line arithmetic conversion. The only ones permitted are:

V and 9

Drifting or non-drifting characters $ S + -

Zero suppression characters Z *

Punctuation characters , . / B

For in-line conversions, pictures with this subset of characters are divided into three types:

Picture type 1: Pictures of all 9s with (optionally) a V and a leading or trailing sign. For example:

'99V999', '99', 'S99V9', '99V+', '$999'

Picture type 2: Pictures with zero suppression characters and (optionally) 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 (optionally) 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, because it has certain characteristics that necessitate a subroutine call. These may be, for instance:

- There is no overlap 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 temp.

- 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', '++V++'

Example: Fixed-Binary to Fixed-Decimal (Compiler Conversion No. 6)

The conversion is performed by converting from binary to decimal via a CVD instruction, with a scale-matching operation (to line up the decimal and binary points) either before or after the CVD (or occasionally both). This scale-matching operation is done by shifts where possible but, depending on scales and precision, a decimal multiplier is sometimes used.

```
DCL SOURCE FIXED BINARY (31,9),
    TARGET FIXED DECIMAL (15,-6);
TARGET=SOURCE;
```

```
L     14,SOURCE
LTR   14,14                     Determination
BNM   Compiler label            Branch if >0
A     14, Constant              Add a constant to negative source
SRA   14,9                      Divide by source scale (2**9)
CVD   14,WKSP+8                 Convert to decimal in workspace
XC    TARGET(3),TARGET          Set zeros in target
MVC   TARGET+3(5),WKSP+8        Transfer value to target
MVN   TARGET+7(1),WKSP+15       Transfer the sign
```

Conversion required	Compiler conversions used
Fixed-decimal to bit-string	No. 7 Fixed-decimal to fixed-binary No. 15 Fixed-binary to bit-string
Floating-point to bit-string	No. 3 Floating-point to fixed-binary No. 15 Fixed-binary to bit-string
Bit-string to fixed-decimal	No. 17 Bit-string to fixed-binary No. 6 Fixed-binary to fixed-decimal
Bit-string to floating-point	No. 17 Bit-string to fixed-binary No. 2 Fixed-binary to floating-point
Character-string to bit-string	No. 10 Character-string to fixed-binary No. 15 Fixed-binary to bit-string
Fixed-binary to character-string	No. 6 Fixed-binary to fixed-decimal No. 12 Fixed-decimal to character-string
Fixed-binary to decimal picture	No. 6 Fixed-binary to fixed-decimal No. 18, 19, or 20 Fixed-decimal to picture
Floating-point to decimal picture	No. 5 Floating-point to fixed-decimal No. 18, 19, or 20 Fixed-decimal to picture
Decimal picture to fixed-binary	No. 21 Picture to fixed-decimal No. 7 Fixed-decimal to fixed-binary
Decimal picture to floating-point	No. 21 Picture to fixed-decimal No. 4 Fixed-decimal to floating-point
Decimal picture to decimal picture	No. 21 Picture to fixed-decimal No. 18, 19, or 20 Fixed-decimal to picture

Figure 10.4. Multiple conversions

MULTIPLE CONVERSIONS

The conversions listed in figure 10.3 can be regarded as fundamental types. A number of other conversions can be performed by using two fundamental conversions in series. These are shown in figure 10.4.

HYBRID CONVERSION

Finally, there is one hybrid conversion that is carried out partially in-line. This is floating-point to character-string, which requires an interpretive routine to analyze the floating-point data (as distinct from the attributes, which all the others use), in order to generate the correct scale factor. This is done by the library, because in-line code would use the same algorithm. However, partial optimization is carried out by setting up a character string of the correct length before calling the library, and then handling the subsequent string assignment in-line.

Raising the Conversion Condition

The PL/I language specifies that when an invalid conversion is attempted on character-string data, the CONVERSION condition will be raised unless CONVERSION has been disabled.

When the CONVERSION condition has been raised, the language allows the program to access the invalid field or character by use of the ONSOURCE or ONCHAR built-in function. The language also stipulates that conversion should be attempted again if an on-unit is entered in which the ONSOURCE or ONCHAR pseudovariable is used to change the invalid field or character.

Raising the CONVERSION condition involves a number of housekeeping problems, which are handled by a special conversion module, IBMBSCV. IBMBSCV is never called by compiled code, since conversions that could raise the CONVERSION condition are not attempted in-line unless the CONVERSION condition is disabled. IBMBSCV produces the correct error code for the error handler, IBMBERR, and looks after the housekeeping problems.

IBMBSCV saves considerable overheads being carried either by all types of errors or by all correct conversions. The reason for the overhead lies principally in the facility offered by the language of using the ONSOURCE and ONCHAR built-in functions to access and optionally change the field causing the error, and subsequently reattempting the conversion on the changed field.

Before any conversion in which the CONVERSION condition could be raised is attempted, the ONSOURCE field in the ONCA must be set up, and the address at which a reattempted conversion should begin must also be placed in the ONCA.

The code carrying out the conversion must then test the validity of the field tc be converted and, if it is invalid, set the ONCHAR field in the ONCA to the first invalid character. The module IBMBSCV is then called to diagnose the conversion and produce the correct error code for the error handler. There are some twenty possible error codes associated with the CONVERSION condition.

If the condition was raised during the execution of stream input, further action is necessary. This is because an on-unit may specify further input, and the buffer which contains the ONSOURCE field may be lost. For example the on-unit might be:

```
ON CONVERSION BEGIN;
ON CONVERSION SYSTEM;  /* PREVENTS
                          RECURSIVE ENTRY*/
   GET LIST (KEYB);
   IF KEYB< 200 THEN ONCHAR ='1';
   ELSE ONCHAR ='9';
   END;
```

If KEYB was in the next record, the source field that caused the conversion would be lost. To prevent this, a VDA is acquired in the LIFO stack, and the source field is stored in this VDA. The ONSOURCE and ONCHAR pointers are altered to point to the field in the VDA, and all further operations are carried out on this field.

The NAB pointer associated with the block in which the interrupt occurred must then be altered so that it encompasses the VDA. The fact that the NAB pointer has been altered must be known in the block for a GOTO out of block to be handled. The reset-NAB bit is accordingly set to one in the relevant DSA. When these operations are complete, IBMBSCV calls the error-handling module IBMBERR.

Chapter 11: Miscellaneous Library Subroutines and System Interfaces

In addition to employing the PL/I libraries for the functions described in previous chapters, the OS PL/I Optimizing Compiler calls on a large number of computational and data-handling subroutines and on subroutines that provide interfaces with the operating system for such functions as TIME and DATE. These miscellaneous library calls are discussed in this chapter. The library subroutines themselves are fully described in the publications IBM System/360 Operating System: PL/I Resident Library Program Logic and IBM System/360 Operating System: PL/I Transient Library Program Logic.

This chapter is divided into two main sections: the first deals with the computational and data-handling subroutines, and the second with miscellaneous system interfaces.

Computational and Data-handling Subroutines

The computational and data-handling subroutines are used to handle all the mathematical built-in functions, the majority of arithmetic built-in functions, and a number of array-handling, structure-handling, and string-handling functions. The extent to which library calls are used depends on the level of optimization specified by the programmer, the type of data involved, and, for string functions, on whether STRINGRANGE and STRINGSIZE are enabled.

ARITHMETIC AND MATHEMATICAL SUBROUTINES

The compiler always uses library subroutines for mathematical functions. The use of compiled code in these circumstances is impracticable. Where possible, arithmetic functions are handled by in-line code. The circumstances in which library subroutines are used for arithmetic functions are listed in figure 11.1.

Considerable use is made of chains of library modules to carry out the various functions. For example, the subroutines that handle complex arithmetic normally call on those that handle real values to process each part of a complex number; similarly, the square-root subroutine is used in the computation of several of the trigonometrical functions.

Arguments are passed to the arithmetic and mathematical subroutines either in registers or in a parameter list addressed from register 1. The use of registers results in faster execution, but allows less flexibility in use of the routines. Compiled code always passes arguments in a parameter list. All built-in functions, except the STRING built-in function, have their arguments passed in a list comprising the addresses of the source and target (and sometimes also the addresses of DEDs). Computational routines are always carried out in floating-point unless otherwise indicated. This may involve conversion before calling the routine.

ARRAY, STRING, AND STRUCTURE SUBROUTINES

A number of array, string, and structure subroutines are included in the OS PL/I Resident Libary. These are used to carry out certain of the array and string built-in functions and a number of other operations. Where possible, in-line code is generated to carry out these functions. However, the enablement of STRINGSIZE, the use of unaligned bit strings, and the use of adjustable and certain varying-length strings will result in calls being made to the library sub-routines.

The subroutines involved in these functions are shown in figure 11.2. Two of them, IBMBAIH and IBMBAMM, are concerned with the handling of data aggregates rather than with the execution of specific operations. They are discussed below.

REAL ARGUMENTS

Function	Data type	Module name	When used
Integer exponentiation	Short floating-point	IBMBMXS	When exponent is a variable
	Long floating-point	IBMBMXL	When exponent is a variable
	Extended floating-point	IBMBMXE	Always
General exponentiation	Short floating-point	IBMBMYS	Always
	Long floating-point	IBMBMYL	Always
	Extended floating-point	IBMBMYE	Always

COMPLEX ARGUMENTS

Function	Data type	Module name	When used
Integer exponentiation	Short floating-point	IBMBMXW	When exponent is a variable
	Long floating-point	IBMBMXY	When exponent is a variable
	Extended floating-point	IBMBMXZ	Always
General exponentiation	Short floating-point	IBMBMYX	Always
	Long floating-point	IBMBMYY	Always
	Extended floating-point	IBMBMYZ	Always

Figure 11.1. Arithmetic operations performed by library subroutines

Handling Interleaved Arrays (IBMBAIH)

IBMBAIH is used to assist the other library array-handling subroutines to process interleaved arrays. It is not called by compiled code.

Interleaved arrays are arrays whose elements are not held contiguously in storage. They occur in arrays of structures. For example, the declaration:

```
DCL 1 STRUCTURE (2),
        2 A(2),
        2 B ;
```

would result in successive storage locations being allocated to elements of A and B as follows:

```
A(1,1),A(1,2),B(1),A(2,1),A(2,2),B(2)
```

Both A and B are interleaved arrays. A is a two-dimensional array, the first row of which is separated from the second by an element of B. As can be seen, the elements of A are not contiguous, nor is there a fixed interval between their addresses.

The interval between the addresses of elements of an interleaved array referred to by varying only the final subscript is always fixed, and these elements can be stepped through by using the last multiplier from the array descriptor. However, such groups of contiguous elements are not themselves necessarily contiguous.

When IBMBAIH is called, it is passed, the number of dimensions in the array, the address of the array descriptor, and the address of a work area in which to construct a table. Basically, IBMBAIH calculates the extent of each dimension and enters this information in the table; it then calculates the increments that must be added in order to step between elements that may be non-contiguous (see figure 11.3). The information in the completed table is used by the calling module to

IBMBAAH	ALL and ANY built-in functions
IBMBAIH	Indexer for interleaved arrays
IBMBAMM	Structure mapping
IBMBANM	STRING built-in function
IBMBAPC	PROD built-in function (fixed-point integer)
IBMBAPF	PROD built-in function (short or long floating-point)
IBMBAPE	PROD built-in function (extended floating-point)
IBMBAPM	STRING pseudo-variable
IBMBASC	SUM built-in function (fixed-point)
IBMBASF	SUM built-in function (short or long floating-point)
IBMBASE	SUM built-in function (extended floating-point)
IBMBAYF	POLY built-in function (short or long floating-point)
IBMBAYE	POLY built-in function (extended floating-point)
IBMBBBA	AND and OR logical operations (aligned bit strings)
IBMBBBC	Compare aligned bit strings
IBMBBBN	Invert aligned bit string (NOT)
IBMBBCI	INDEX built-in function (character string)
IBMBBCK	Concatenate character strings and REPEAT built-in function
IBMBBCT	TRANSLATE built-in function (character string)
IBMBBCV	VERIFY built-in function (character string)
IBMBBGB	BOOL built-in function
IBMBBGC	Compare unaligned bit strings
IBMBBGF	Bit-string assignment (aligned, source and target)
IBMBBGI	INDEX built-in function (bit string)
IBMBBGK	Concatenate bit strings, REPEAT built-in function, and assign
IBMBBGS	Produces SLD (SUBSTR built-in function)
IBMBBGT	TRANSLATE built-in function (bit string)
IBMBBGV	VERIFY built-in function (bit string)

Figure 11.2. Array, structure, and string subroutines

address successive elements of the array using simple code.

Structure Mapping (IBMBAMM)

Structures are normally mapped during compilation. However, certain structures that contain adjustable strings or arrays cannot be mapped until the actual lengths or bounds are known. Compiled code calls on the module IBMBAMM to carry out this mapping. There are four entry points:

IBMBAMMA Compute length of structure.

IBMBAMMB Map structure in PL/I manner.

IBMBAMMC Map structure in COBOL manner (for interlanguage comunication or for files declared with the COBOL option).

IBMBAMMD Map structure declared with REFER option.

Miscellaneous System Interfaces

In addition to the system interface used for input and output, the PL/I Optimizing Compiler makes use of a number of other system facilities. These are for the DELAY, DISPLAY, and WAIT statements, the TIME and DATE built-in functions, and the sort/merge and checkpoint/restart built-in subroutines.

Calls to these facilities are made through library subroutines held in the OS PL/I Resident Library. These subroutines act as an interface, issuing any SVC calls that may be necessary, and handling housekeeping problems. The descriptions of the subroutines in this chapter are kept to a minimum except where the housekeeping problems are large and have a major effect on the contents of main storage. In these cases, background information is given and the various control blocks are explained, thus enabling the situation during execution to be understood.

The OS macro instructions referred to below are described in IBM System/360 Operating System: Supervisor and Data Management Macro Instructions.

TIME

The PL/I TIME built-in function is implemented by issuing a GETIME macro instruction. This is done by the module IBMBJTT.

On entry from compiled code, register 1

points to the address of the character-string target. The TIME macro instruction is issued using the TU parameter. The time is returned in units of 26.04 microseconds and the module converts this into PL/I defined format 'hhmmssttt' in character format.

DATE

The PL/I DATE built-in function is implemented by module IBMBJDT.

On entry from compiled code, register 1 points to the address of the date character string. The TIME macro instruction is issued. On return register 1 contains the date in yydddc packed decimal format. The year is placed in the target character string in character form. The day of the year is then compared against a table indicating the number of days in each month. If the year is a leap year the number of days for February is set to 29 in the table. The days and months are then set in the character string and the result returned to compiled code in the form yymmdd.

DELAY

The PL/I DELAY statement is implemented by calling the DELAY module IBMBJDY. Register 1 is pointed at the milliseconds delay required. The milliseconds are converted into units of 26 microseconds and the result stored in a fullword addressed by the TUINTVL parameter in the STIMER macro instruction. The STIMER macro instruction is then activated and the delay started. After the delay control is returned to the calling program.

DISPLAY

The PL/I DISPLAY statement is implemented by the module IBMBJDS. There are two entry points:

1. IBMBJDSA - entry from compiled code.
2. IBMBJDSB - entry from IBMBJWT or IBMTJWT when a WAIT for the EVENT is reached.

If the parameter list passed to the module has one element, then the entry is for DISPLAY only, and a VDA is obtained. If there are two parameters, then the entry is for DISPLAY REPLY and a VDA is again obtained. If there are three parameters, then the entry is for DISPLAY REPLY EVENT. If the event variable is active ERROR is raised. If the event variable is inactive it is set active, I/O display and incomplete, and non-LIFO storage is obtained in which to build the parameter list.

Next the reply buffer, if present, is filled with blanks and, if the reply string is variable length, its current length is set to the maximum length. The parameter list to the WTO macro is now built in the storage obtained, the address of the ECB put into the event variable if there is one, and a WTO macro issued. Finally, if DISPLAY REPLY without EVENT was specified, a WAIT macro is issued for the ECB. Return is then made to compiled code.

SORT/MERGE

The PL/I programmer can make use of the OS sort/merge facilities through a call to the built-in subroutine PLISORT. The method of using the facility is fully described in the publication IBM System/360 Operating System: PL/I Optimizing Compiler Programmers' Guide.

The OS sort/merge program includes a number of user exits that can be conveniently thought of as allowing the programmer to write sections of code that become included in the sort/merge routines. Two of these user exits can be used by the PL/I programmer: user exit 15 allows records to be set up by PL/I and passed to the SORT routines; user exit 35 allows records that have been sorted to be passed to and processed by the PL/I program.

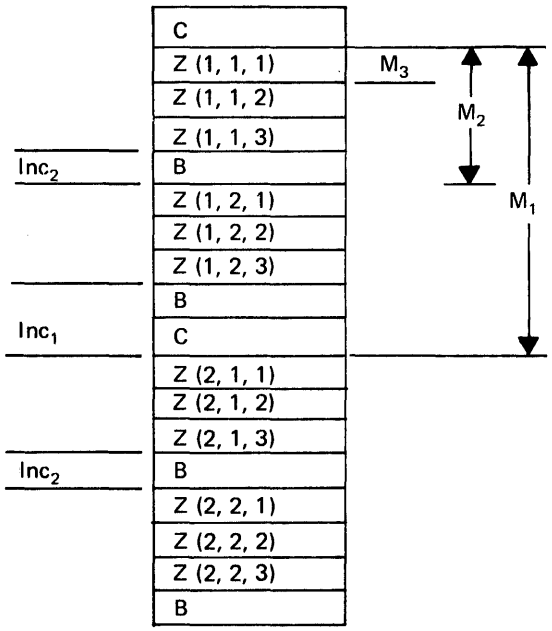
Exits are not allowed in the PL/I language. To overcome this problem, code is inserted between the sort/merge modules and the PL/I routines. A bootstrap module, IBMBKST, is used, and this module acts as an interface between SORT and PL/I. The bootstrap module saves the PL/I environment and restores it on return from sort/merge so that the PL/I exit-15 or exit-35 code can operate in a PL/I environment. Similarly, the bootstrap module restores the environment for SORT on return from the exit.

Saving and restoring the environment consists of replacing the address of the error handler in the TCA with the address of an error routine in IBMBKST, and vice-versa.

Declaration

```
DECLARE   1 X(2),
            2 C,
            2 Y (2),
              3 Z (3),
              3 B;
```

Storage



Z is a three-dimensional interleaved array, for which

M_1, M_2, and M_3 = multipliers held in array descriptor (see chapter 4)

Inc_1 and Inc_2 = intervals between addresses of successive elements of Z when subscripts for first and second dimensions, respectively, change

The increment when the subscript for the ith dimension changes is computed as follows:

$Inc_i = M_i - E_{i+1} * M_{i+1} + Inc_{i+1}$

Where E_{i+1} is the extent of the (i+1)th dimension.

Increment table for array Z (as initialized by IBMBAIH)

2nd dimension	2	subscript count
	2	extent of dimension
	Inc_2	increment
1st dimension	2	subscript count
	2	extent of dimension
	Inc_1	increment

Note: IBMBAIH returns the extent of the nth dimension in register 1. (In this example, the extent of the 3rd dimension = 3.)

Figure 11.3. Indexing interleaved arrays

Housekeeping Problems

Various housekeeping problems occur in the user exit procedures, since there is no DSA chain through the SORT modules. Particularly difficult is the handling of a GOTO out of the exit procedure that passes control to a procedure that was activated before the procedure that originally called the sort program. This action implicitly terminates SORT. However, SORT will not be terminated by standard PL/I action, since it does not function in the PL/I environment.

The problems are overcome by setting up a chainback that omits the SORT DSAs and includes a DSA that is specially flagged so that it can be recognized by the GOTO code. The chaining of save areas in shown in figure 11.4.

When IBMBKST is called an area of workspace is acquired by the bootstrap routine IBMBKST. This consists of one level of library workspace, which is flagged and chained to look like two DSAs.

If the SORT program is terminated by a GOTO out of the block that contains the PL/I exit program, the SORT routine has to be terminated before the GOTO can be completed. This is done by the GOTO routine looking for a specially flagged DSA in the chain. This is the second save area of IBMBKST. If one is found, a return code of 8 is set up and return made to the SORT routine. If there is a GOTO or an error, then error code 16 is set instead of 8 if the SORT program product being used is that which supports this return code to exits. This results in the termination of the SORT routine, and the GOTO can then be continued in the usual manner by following the DSA backchain through the bootstrap routine until the target DSA is reached.

For handling on-units in the exit procedure, the DSA chain can be followed without reference to SORT.

Restoration of the PL/I Environment on Exit from SORT

When an exit is made from SORT, it is necessary to restore the PL/I environment. The method used is to have code that restores the registers at the point to which SORT makes its exit. Use is made of the SORT exit table shown in figure 11.4. Whichever exit is taken, control passes to this code. The code saves the registers passed by SORT and restores the registers of the bootstrap module IBMBKST, thus restoring the PL/I environment. The save area of the SORT bootstrap routine is addressed by means of an offset from the code that is being executed. This is possible because the SORT exit table and the register save area are both held in the same workspace at a fixed offset from each other. The code is not included in the bootstrap module, in order to preserve reentrancy.

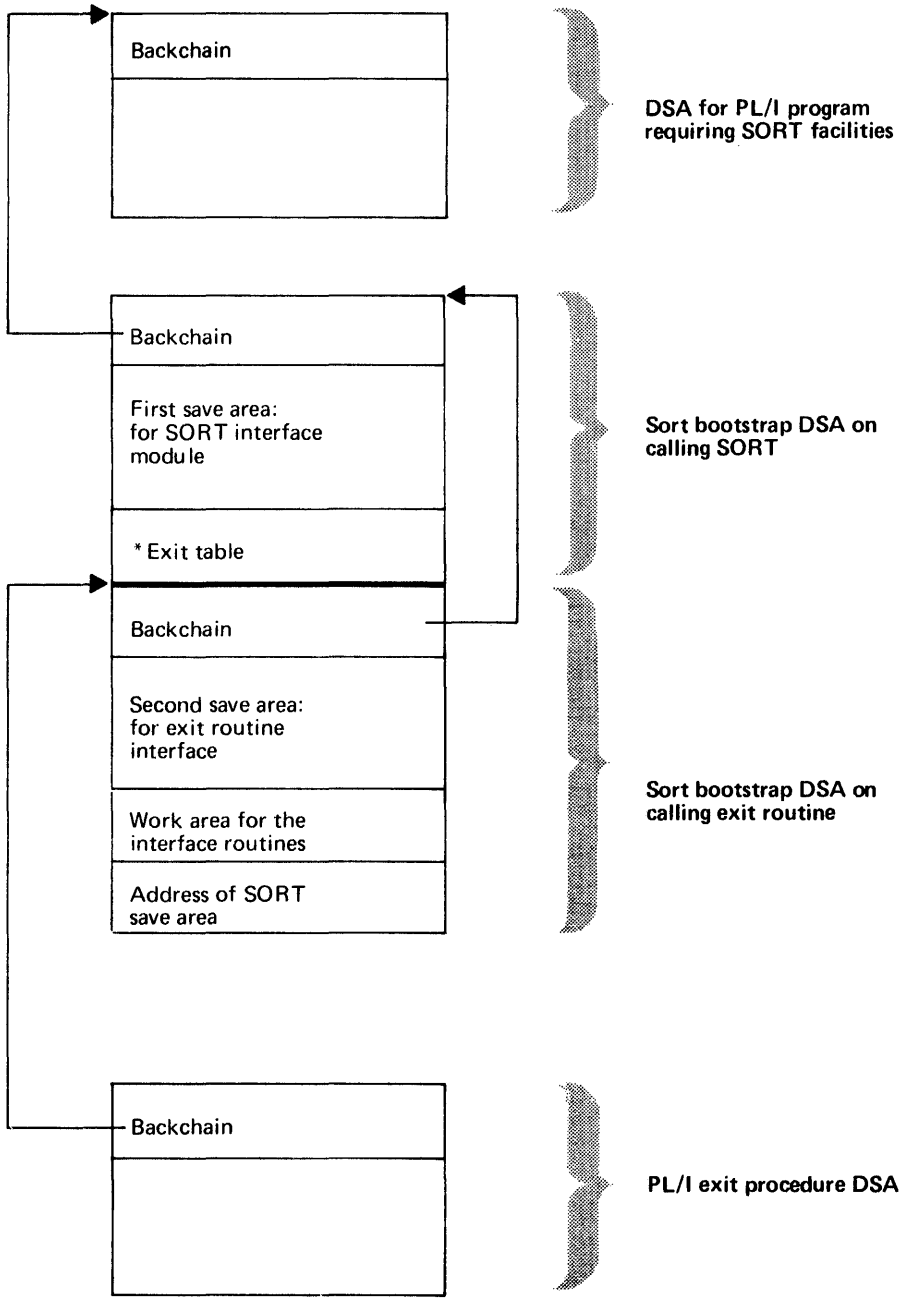
If there is an error in SORT, control is also passed to code which restores the environment, and passes control to IBMBKST and then to IBMBERR.

Summary of Work Done by the SORT Module

Before calling the SORT program, IBMBKST does the following:

1. Obtains a VDA for two DSAs.
2. Creates a parameter list suitable for SORT.
3. Sets up addressability code for exit routine, if any.
4. Changes the interrupt handler address so that an interrupt results in entry being made to a section of the sort bootstrap. The sort bootstrap then determines the error, puts out a message to SYSPRINT indicating that a program check has occurred during the execution of SORT, and then terminates the program.

When a SORT E35 or E15 exit is being taken, the addressability code saves the registers of SORT and reestablishes the PL/I environment, and then branches to an entry point of IBMBKST, which:



*Exit table

	NOP	0	not used
Entry point for E15	BC	15,12(15)	branch to exit code for E15 exit
Entry point for E35	BC	15,12(15)	branch to exit code for E35 exit
	STM	14,12,12(13)	save sort registers
	L	2,28(15)	locate bootstrap save area
	LM	2,12,28(2)	restore bootstrap registers
	B	exit bootstrap	initialized address of routine
	DC	A (save area 1)	address of first save area

Figure 11.4. DSA chaining during the execution of SORT

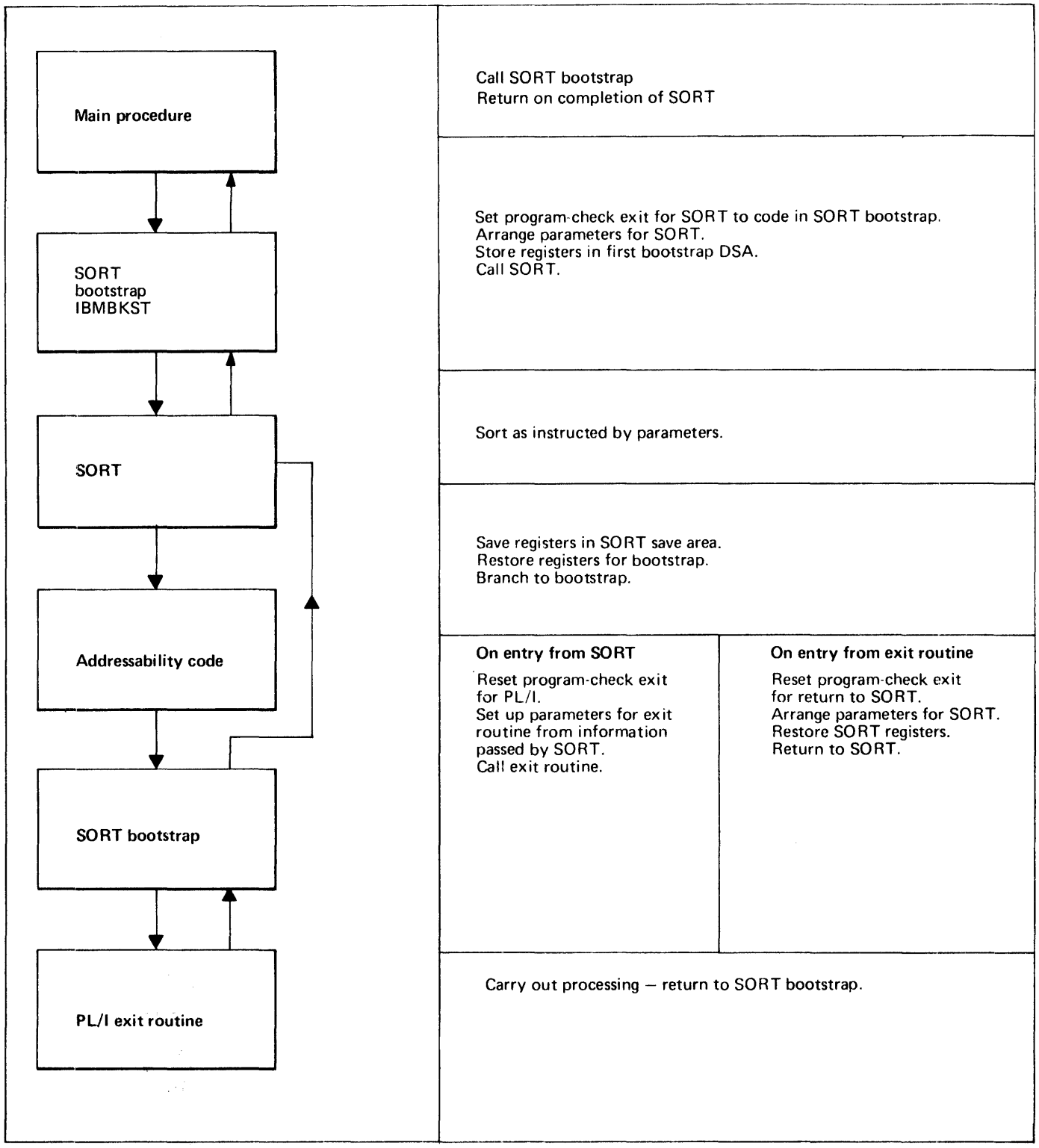


Figure 11.5. Summary of action during use of a SORT exit

1. Restores the PL/I interrupt handler address, so that control will pass to the PL/I error-handling routines if a program interrupt occurs.

2. Sets up parameters for the PL/I exit routine from information passed by SORT.

3. Calls the PL/I exit routine.

Setting the return code in the PL/I exit program resets the parameters that IBMBKST passes to the SORT routines.

Storage for SORT

Storage for sort/merge workspace and the modules used is obtained in the LIFO stack. A VDA of the correct length is obtained by the bootstrap module. The length required must be specified in the arguments that are given in the call to PLISORT. These actions are summarized in figure 11.5.

CHECKPOINT/RESTART

The PL/I Optimizing Compiler allows the programmer to make use of the system checkpoint/restart facilities by calling the built-in subroutine PLICKPT. This is implemented by a call to the resident-library subroutine IBMBKCP, which issues the CHKPT macro instruction.

Before the CHKPT macro instruction is issued, two control blocks must be set up. One of these control blocks contains the names of all tape files that are open; it is used to reposition the tapes on restart. The other control block contains verification information for all disk files that are open; it is used to verify that the disk packs are on the same devices on restart as they were when the check-point was taken. The two control blocks are held in workspace acquired by the module IBMBKCP.

When a restart is made, control is passed to the module IBMBKCP at a fixed entry point. After carrying out necessary checks, control is then returned to the calling routine in the normal manner. Control is thus returned to the statement after the call to PLICKPT, and processing continues.

WAIT

The PL/I WAIT statement allows the programmer to specify that processing shall halt until a specified number of events are complete. In the OS PL/I Optimizing Compiler, an event can be associated with either a record I/O operation or a DISPLAY statement, or it can be an inactive event that is not associated with any operation.

All information relating to an event is kept in an event variable. This is a control block of five words in length; it is treated for storage allocation like any other PL/I variable. The event variable holds information on whether the event is associated with an operation and whether it is complete; it also records the status of the event (i.e., whether the associated operation was completed successfully or otherwise). When an event is associated with an operation, it is said to be active; otherwise, it is said to be inactive.

When the wait statement is used, the keyword WAIT is followed by a list of events that are to be waited on. A number can follow this list, indicating that only that number of events need be completed before processing can continue. Typical WAIT statements are:

```
WAIT (EVENT1,EVENT2);

WAIT (EVENT1,EVENT2) (1);
```

For the first statement, both the events would have to be completed before processing could continue. For the second statement, processing would continue as soon as either of the events was complete.

Event Variables

When storage is allocated for an event variable, the event variable is set inactive and incomplete. When the EVENT option is used to associate the event with an operation, the event variable is set active and incomplete. When a WAIT statement is executed and the operation associated with the event has been completed, the event variable is set inactive and complete. The status of the event is also set at this time, indicating whether or not the operation was successfully completed.

The PL/I language allows the programmer to set complete or incomplete any event, by use of the COMPLETION pseudo-variable. This sets the appropriate bit in the event variable. The completion status may be

```
WAITER: PROC OPTIONS (MAIN);

          ON TRANSMIT (A) CALL L;
          ON TRANSMIT (C) CALL L;
          ON TRANSMIT (X) CALL L;

          ON RECORD (A) CALL M;
          ON RECORD (C) CALL M;
          ON RECORD (X) CALL M;
          K=0;
1         READ FILE (A) INTO (B) EVENT
         (E1);
2         READ FILE (C) INTO (D) EVENT
         (E2);
           •
           •
           •
3          WAIT (E1,E2);
           •
           •
           •
4          IF K=1 THEN WAIT (E2);
           •
           •
           •
5 BOOTLE:  WAIT (E3);
           •
           •
           •
       L: PROC;
6          COMPLETION (E3)='1'B;
7          GO TO BOOTLE;
           END L;
       M: PROC;
8          COMPLETION (E3)='1'B;
9          WAIT (E2);
10         K=1;
11          READ FILE(X) INTO(Y) EVENT
              (E2);
           END M;

           END WAITER;
```

Figure 11.6. Example of WAIT implementation problems

inspected by means of the COMPLETION built-in function. The PL/I language also allows the programmer to inspect and change the status of an event, by means of the STATUS built-in function and pseudo-variable.

WAIT Statement (Non-Multitasking)

The WAIT statement in a non-multitasking environment is implemented by a call to the resident library routine IBMBJWT. IBMBJWT is passed a set of parameters consisting of the addresses of the event variables and the number of events that have to be completed. If the number of events that have to be completed is not specified, all the events in the list must be completed. (For the multitasking situation, see chapter 14.)

The WAIT makes use of the OS data-management WAIT macro instruction. However, because of the differences between the facilities offered by the OS and the PL/I language, considerable housekeeping problems are involved for waits on more than one event. For waits on single events, the problems are small and are described at the end of this section.

When a WAIT or associated macro instruction is issued to the OS supervisor, the event is considered to be complete when input/output transmission is finished. In PL/I, however, a WAIT statement is not considered complete until any error-handling activity caused by the operation which was being waited on is finished. The error handling may include entry into an on-unit, and further WAIT statements may be executed in the on-unit. This process can continue to any number of levels of interrupt.

PL/I also allows the programmer direct control over the completion of an event by use of the COMPLETION pseudo-variable. Consequently, the PL/I programmer need not associate an event variable with an input/output operation, but can use it instead as a flag, setting the event complete at any point in the program.

WAIT or associated macro instructions issued to the supervisor are completed by setting a completion bit in the ECB (event control block) which is held in the IOB. At the PL/I level, completion is indicated by setting the completion bit in the event variable. Thus a WAIT operation is carried on at two levels, the PL/I level and the system level.

Housekeeping Problems

The problems involved in implementing the WAIT statement may be illustrated with examples from the skeleton program in figure 11.6. Four problems arise. They are:

Problem 1: If an event being waited on in a multiple WAIT statement is completed in an on-unit entered while processing one of the other events in the statement, this must be made known to the first WAIT statement. Setting the event variable complete is not sufficient, because the event variable may be used again during the

on-unit. Suppose that the RECORD condition is raised during the execution of the WAIT statement numbered 3 in figure 11.6, for the operation associated with event E1. The following then takes place:

1. Control passes to procedure M.

2. The statement WAIT(E2) is then encountered, and the program waits until event E2 is completed. When this occurs, the event variable is set complete and inactive.

3. Event E2 is then used in a further I/O operation (statement 11), causing the event variable to be set active and incomplete.

On return to the main program, there would be no way of determining from the event variable for E2 that the original event E2 had been completed. The problem is solved by the use of control blocks called event tables (EVTABs). An EVTAB is set up by the wait module each time a WAIT statement is encountered; it contains entries for each incomplete event specified in the statement. The entries are termed EVTAB elements. Each element is chained to its corresponding event variable and contains a bit that can be set to indicate that the event has been completed. In the above example, therefore, EVTAB elements for E1 and E2 are set up when the wait module is called at statement 3. When the on-unit is entered, the WAIT statement 9 causes a further EVTAB to be set up with an entry for E2. The event variable pointer is reset to address the latest EVTAB elements, and a field in this element is set to point to the previous EVTAB element for E2. When event E2 is completed (without causing any I/O conditions to be raised), the event variable and each EVTAB element for E2 is set complete and inactive, and a bit in the event variable is set to indicate that the chain of EVTAB elements is no longer associated with the event variable. When statement 11 is executed, the event variable is set active and incomplete. After the on-unit has been executed, the wait module sets the EVTAB element and event variable for E1 complete and inactive. It then tests any remaining EVTAB elements to determine whether they were set complete during an on-unit; in this case, it finds that the next EVTAB element (for E2) has been set complete and that there are no more events to process. Execution therefore continues until statement 4 is executed, at which time a new EVTAB element is created for E2 and chained to its event variable.

Problem 2: A method must be provided to signal that an event waited on in an on-unit is already being waited on in the procedure that caused entry to the on-unit. Suppose that the RECORD condition is encountered in the operation associated with E2 (statement number 2) during processing of the WAIT at statement number 3. The following then takes place:

1. Control passes to procedure M.

2. A further WAIT on E2 is encountered (statement number 9). Since E2 cannct now be completed, a mechanism must be available to raise the ERROR condition; otherwise, the program would never get out of the wait state,

The problem is solved by setting a flag in the event variable whenever an on-unit is entered during WAIT statement processing. If the wait module is subsequently reentered from an on-unit, to process a WAIT on the same event, it finds that this bit is set and raises the ERROR condition.

Problem 3: If there is a GOTO out of an on-unit, this involves setting an event variable complete, and terminating the WAIT statement. Suppose the TRANSMIT condition is raised during the WAIT statement numbered 3, 4, or 9. The procedure L is entered and the following takes place:

1. E3, which is a dummy event, is set complete.

2. A GOTO is executed to the label BOOTLE.

If no other action were taken, the event that caused entry to the on-unit (either E1 or E2) would not be set complete; any subsequent WAIT on that event would thus cause the wait module to be invoked, with unpredictable results. The problem is solved by setting a flag bit in the current DSA whenever the wait module is called. (The method is similar to that used to cater for a GOTO out of a SORT exit, and uses the same flag bit.) If the GOTO module finds that the bit is set, it returns to the wait module; the wait module sets the event variable complete and inactive and then returns to the GOTO module to continue the GOTO out of the on-unit. Only the event that caused entry to the on-unit is set complete. Any other incomplete events specified in the WAIT statement are left incomplete.

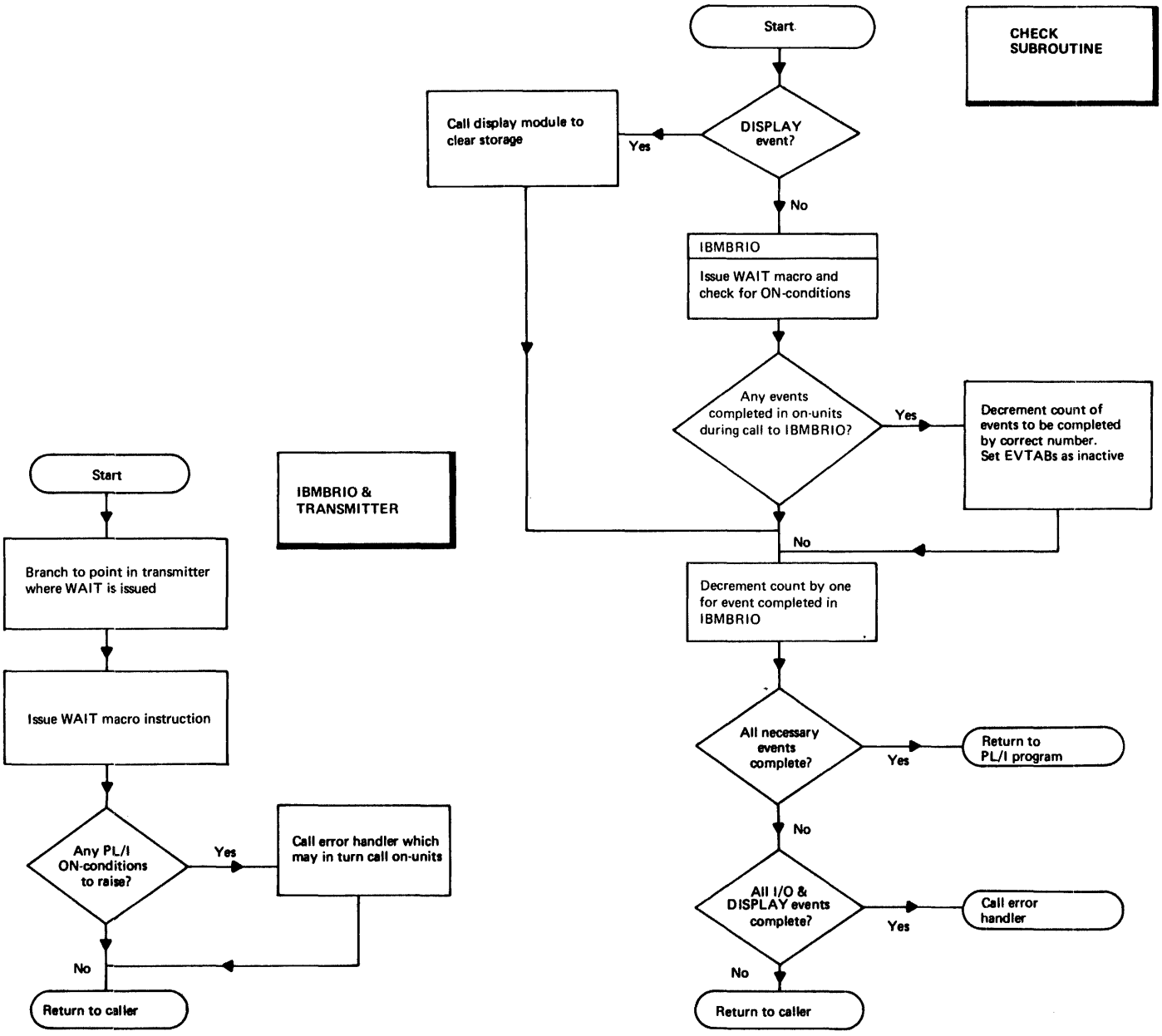


Figure 11.7. (Part 1 of 2). Summary of the wait statement

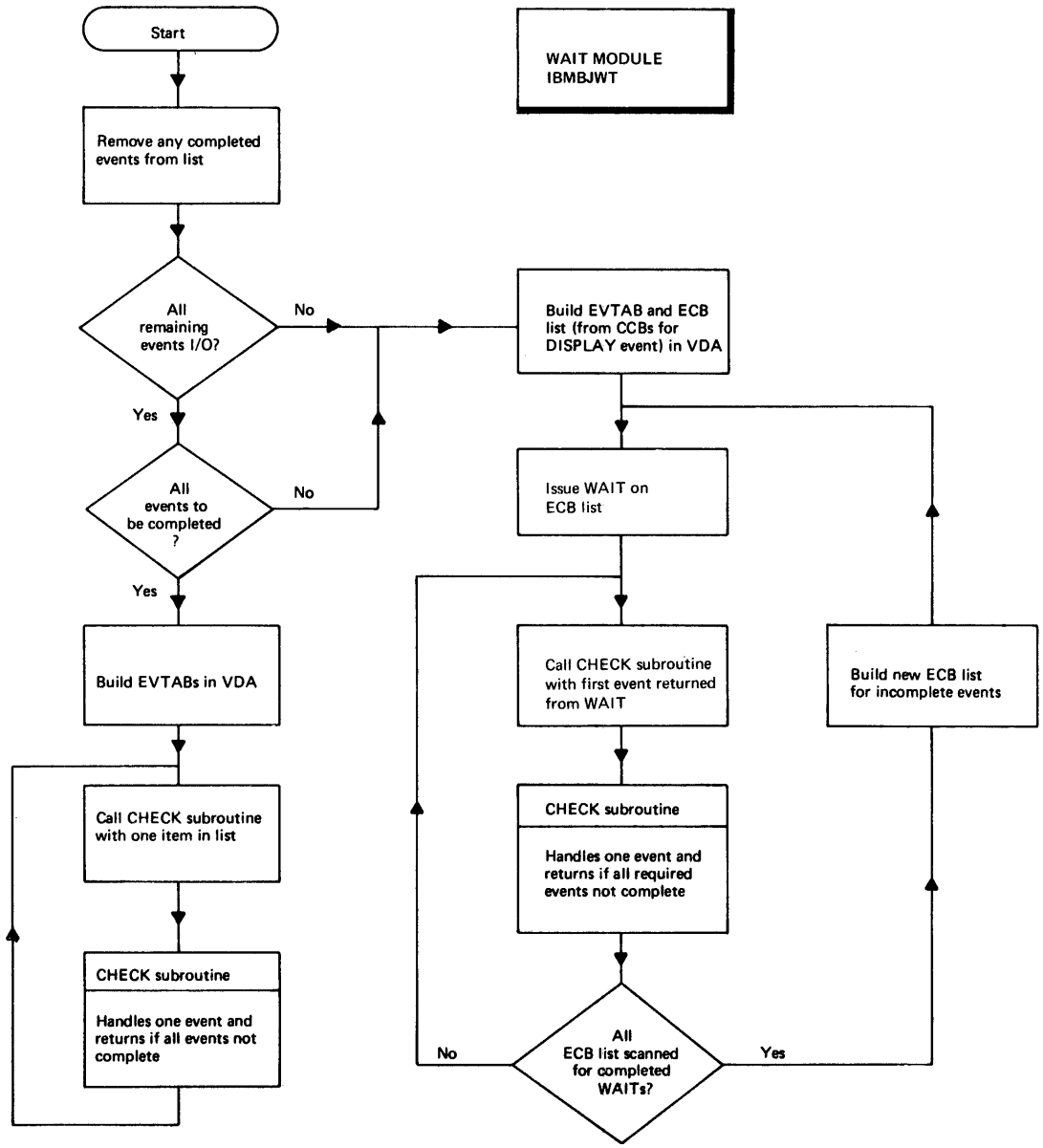


Figure 11.7. (Part 2 of 2). Summary of the wait statement

Problem 4: If control reaches label BOOTLE without the TRANSMIT or RECORD condition having been raised, the event E3 can never be completed. Some method must be available of making this fact known, otherwise the program would go into an indefinite wait on an event that could never be completed. This problem is solved by setting an event variable active only when it is associated with an operation. Thus, if a WAIT statement specifies an event that is inactive and incomplete, the wait module causes the program to be terminated. (If a WAIT statement specifies more than one event and one of the events is inactive and incomplete, the program is not terminated immediately because it is possible, although unlikely, that the incomplete event will be completed by the COMPLETION pseudovariable in an on-unit entered as a result of an I/O condition raised while processing one of the other events specified in the WAIT statement.)

Control Blocks

Four control blocks are involved in the implementation of the WAIT statement. These are shown in detail in appendix A.

1. Event variable. Used to hold all information about the event at a PL/I level. Fields indicate whether it is active or inactive; complete or incomplete; whether it is already being waited on at a previous interrupt level; the type of operation with which it is associated. Each event variable contains the address of its associated ECB or CCB and, if it associated with an I/O event, the address of the FCB for the file.

2. ECB (event control block). Used to hold information about the event at the system level. For I/O events, ECBs are part of the IOB. For DISPLAY events, the equivalent control block is the display control block, which is set up by the display module.

3. EVTAB (event table). Created for each entry to the WAIT module; comprises an element for every incomplete event that is to be waited on. The EVTAB is held in a VDA acquired by the WAIT module.

4. ECB list. This is a list of ECB addresses that is created in circumstances that are explained below. The ECB list is held in the VDA described above, and acts as an argument list for the WAIT macro instruction.

Wait Module (IBMBJWT)

The actions of the wait module, IBMBJWT, are shown in the flowchart in figure 11.7, and are described in detail in the publication OS PL/I Resident Library Program Logic.

As the flowchart shows, the WAIT module sometimes issues a WAIT macro instruction, and sometimes relies on the CHECK macro instructions in the PL/I transmitters. The reasons for this are as follows.

The CHECK macro instruction in the transmitter can only be used for I/O events, and only one transmitter can be called at a time. If only a certain number of the events in an event list need to be completed, it is uneconomic to pass these events one at a time to the transmitter, because the first event passed could be the last to finish. Consequently, whenever non-I/O events are involved and whenever less than the total number of events in an event list have to be completed, an ECB list is generated for all incomplete events and a WAIT macro instruction is issued.

The WAIT macro instruction returns control as soon as any event in the list is complete, thus allowing an event list to be handled efficiently when only a number of events have to be completed. For I/O events, it is still necessary to issue the CHECK macro instruction in the transmitter, even though the events are known to be complete. This is because the CHECK macro instruction carries out various checking functions as well as waiting until the event is complete.

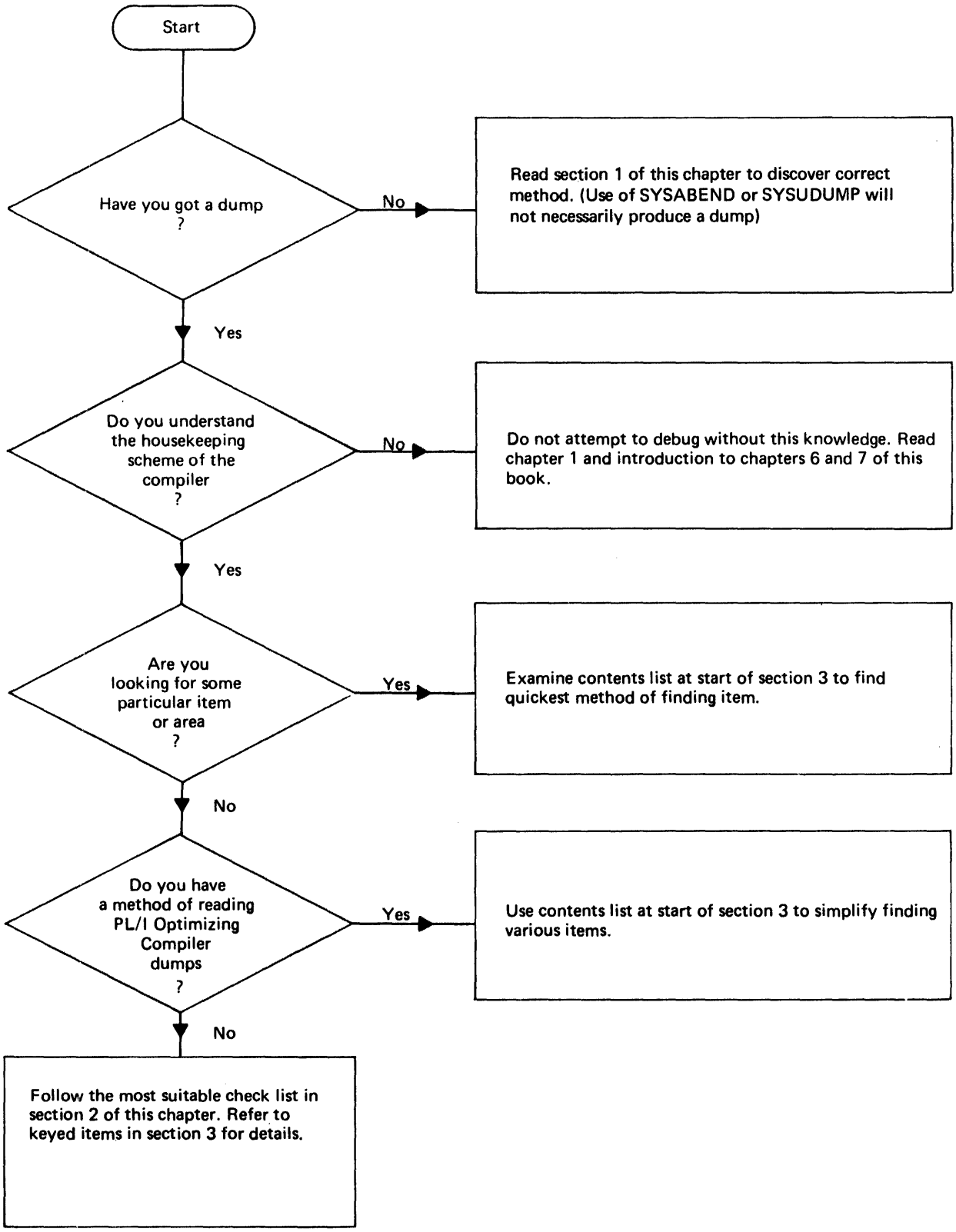


Figure 12.1. How to use this chapter when debugging

Chapter 12: Debugging using Dumps

The OS PL/I Optimizing Compiler allows the programmer to obtain an execution time dump only by calling PLIDUMP. Using SYSABEND or SYSUDUMP in the JCL will not result in a dump after a program interrupt or, except in certain exceptional cases, after an ABEND. This is because the program interrupt exit and the task asynchronous exit are reset by issuing SPIE and STAE macro instructions in the program initialization routine. These SPIE and STAE routines result in all interrupts, and the majority of ABENDs, being passed to the PL/I error handler.

Certain types of program error can, however, result in overwriting of the control information used by the error handling routines. When this occurs an ABEND will be issued that results in system action. This ABEND has a user code of 4000. Provided that a SYSABEND or SYSUDUMP DD card was included in the JCL an ABEND dump will then be generated.

ABEND dumps are issued in three circumstances.

1. When an interrupt occurs during the execution of one of the error handling routines.

2. When housekeeping control blocks have been overwritten after an ABEND in the program.

3. If the NOSPIE or NOSTAE option has been used, and the action inherited by the PL/I program is to give a dump.

The first two of these situations are most probably caused by overwriting of control information by the PL/I program. The first can be identified because a message is sent to the console that reads 'Interrupt in error handling routines program terminated', and the ABEND code will be 4000.

Chapter 7, 'Error Handling' describes the methods used to handle interrupts and ABENDs. It also describes the implementation of PLIDUMP. This chapter is concerned solely with debugging using the facilities provided.

It is always possible for the programmer to ask an operator to take a stand-alone dump at any point in the program. The need to do this should, however, occur only infrequently.

How to use this Chapter

This chapter contains information on how to obtain and interpret dumps, and on how to identify compiled code, data, and control blocks within a dump. Some knowledge of the compiler's housekeeping scheme, described in other chapters of this book, is assumed. Trying to use a dump without this knowledge can result in a great deal of wasted time. To acquire a quick overall picture, chapter 1 and the introduction to chapters 6 and 7 should be read. A summary of how to use this chapter when debugging is given in figure 12.1.

This chapter is divided into four sections:

Section 1: How to obtain a PL/I dump

Section 2: Recommended debugging procedures

Section 3: Locating specific information

Section 4: Special considerations for multitasking

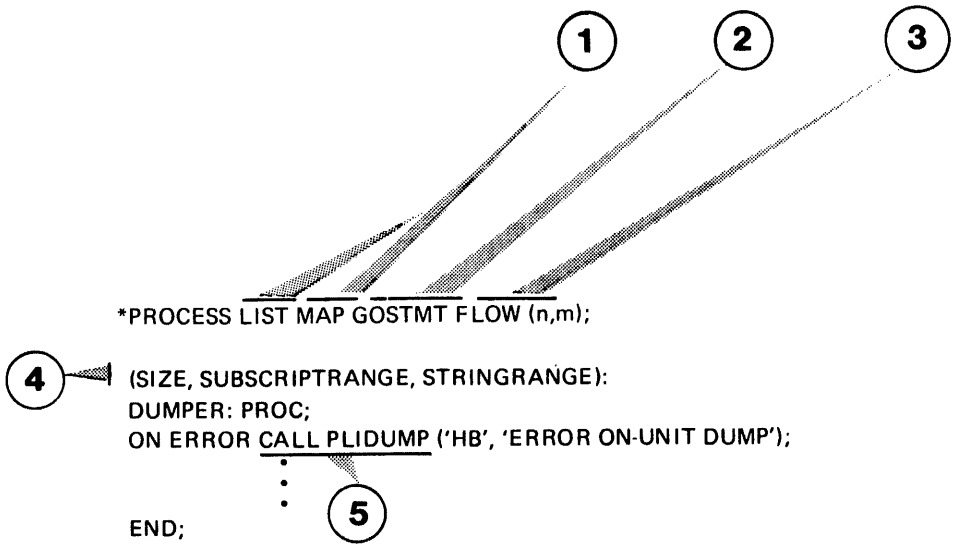
Section 1 explains how to obtain a hexadecimal dump of a PL/I program. It also gives some suggestions on the use of various compiler and PL/I options that may prove useful when debugging.

Section 2 offers two recommended courses for debugging a PL/I program by use of a dump. The first course deals with a PL/I dump that has been called from an ERROR on-unit and is being used to debug the problem program. The second course deals with the situation in which an OS system dump has been generated, probably because the housekeeping control blocks have been overwritten.

Section 3 describes how to find various data areas and other information. It is indexed and numbered for quick reference.

Section 4 describes the special considerations that must be taken into account when debugging a program that uses multitasking.

Before taking a dump, Section 1 should be read, because the methods used are not those familiar to programmers using OS. Sections 2 and 3 are for use when debugging. Programmers who know what they



1. These options give compiled code listing and static storage map, essential for interpreting any dump.

2. Permits trace of statement numbers in original source program, and simplifies program checking.

3. Provides trace of last n branch-out/branch-in points in up to m blocks if SNAP or PLIDUMP with trace is used. MAP results in the generation of a table showing offsets of static and automatic variables from their defining bases.

4. Prefix options. The use of these PL/I checkout options is strongly urged. Since, however, they cause an increase both in the size of object code and in execution time, it may be necessary to restrict their use to suspected blocks or statements.

5. Two arguments can be passed to PLIDUMP. They are the dump options character string and the dump identifier. The format of the call statement is:

CALL PLIDUMP (character-string-expression 1, character-string-expression 2);

Dump options character string
(Default is 'TFC')

T	Trace information required
NT	No trace information required
F	File information required
NF	No file information required
S	Stop after dump
C	Continue after dump
H	Hexadecimal information required
NH	No hexadecimal information required
B	Control block information required
NB	No control block information required
A	Dump all tasks
O	Dump current task only
E	Exit from task after dump

Dump identifier character string

Printed at head of dump. May be up to 90 characters long.

Figure 12.2. Code for debugging

are looking for should refer directly to the contents table in section 3. This will direct them to numbered sections which give details of how to find particular items. Programmers who have no preferred scheme of their own can follow the recommended procedures in section 2. Section 2 crossrefers to the items in section 3, so that the details of the steps involved may be quickly found.

Section 1: How to Obtain a PL/I Dump

In order to get a formatted PL/I dump, the programmer must include a call to PLIDUMP in his program.

CALL PLIDUMP

The statement CALL PLIDUMP may appear wherever a CALL statement may legitimately be used. It has the following form:

```
CALL PLIDUMP
   (character-string-expression 1,
   character-string-expression 2);
```

Character-string-expression 1 is a "dump options" character string consisting of one or more of the following dump option characters:

T Trace. A calling trace through all active DSAs is generated. When an on-unit DSA is encountered, the values of the relevant condition built-in functions are given. The reason for the entry to the on-unit is also given if the ERROR or FINISH conditions are raised as standard system action for another condition.

NT No trace. A calling trace is not given.

F File information. A complete set of attributes for all open files is given, plus the contents of all accessible buffers.

NF No file information required.

S Stop. The program will be terminated after the dump.

C Continue. Execution of the program will be continued after the dump.

H Hexadecimal. A SNAP hexadecimal dump of the partition will be given. If trace information is requested, the TCA and DSA addresses will be given. If file information is requested, the addresses of the FCBs will be given and the contents of all accessible buffers will be printed in hexadecimal notation as well as in character.

NH No hexadecimal dump required.

B Blocks. The contents of the TCA, TIA, DSAs, FCBs, and file buffers are printed in hexadecimal notation.

NB No block information required.

Tasking Options

A All

All results in a dump of all active tasks including the control task - see chapter 14.

O Only

Only results in a dump of the current task and a dump of the control task.

E Exit

Exit results in the termination of the task after the dump.

The default options are TFCANHNB. That is, trace information, file information, no block information, no hexadecimal dump, all tasks, and continuation after the information has been put out.

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

Character-string-expression 2 is a "user identifier" character string of up to 90 characters chosen by the PL/I programmer. It is printed at the head of the dump. If the character string is omitted, nothing is printed.

If PLIDUMP is called a number of times in a program a different user identifier should be used on each occassion. This will simplify identification of the point at which the dump was called.

RECOMMENDED CODING

For PLIDUMP to produce a dump, a DD card for PLIDUMP must be included in the JCL. PLIDUMP can be called from anywhere in a program, but the normal method used when debugging will be to call PLIDUMP from an on-unit. As continuation after the dump is one of the options available, PLIDUMP can

be used as a snap dump to get a series of dumps of main storage throughout the running of the program.

By including the statement CALL PLIDUMP ('HB','dump identifier'); in an ERROR on-unit, it is possible to obtain a hexadecimal dump, with control blocks identified and formatted, should an error occur. If an ERROR on-unit is being included in a program, care should be taken that there are no further ON ERROR statements which might override the on-unit requesting a dump.

Suggested code for use when debugging with a dump is given in figure 12.2.

AVOIDING RE-COMPILATION

If an ERROR on-unit containing a call to PLIDUMP is to be included in an existing program, it is necessary to re-compile the program. This course is advisable as it allows other diagnostic aids, such as SUBSCRIPTRANGE, to be included. However, if re-compilation is not desirable, a PL/I dump can be obtained by using a small bootstrap routine that contains an ERROR on-unit calling PLIDUMP. This routine can be compiled and then link-edited with the object module of the program that needs to be dumped. The on-unit will then be inherited by the program that requires a dump, and a dump will be generated when an error occurs. A suitable bootstrap program is shown in figure 12.3. When using this method, the bootstrap must be link-edited as the MAIN procedure; it should therefore be passed to the linkage editor before the program that requires dumping, since that program will also have the MAIN option. If the program that requires dumping expects to be passed parameters, the bootstrap procedure should use an identical parameter list in its PROCEDURE statement, and should include an identical argument list in the CALL statement used to invoke the inner procedure.

If the program that requires dumping already has an ERROR on-unit, this will override the ERROR on-unit in the bootstrap program.

In certain circumstances, a dump can still be obtained.

1. If the reason for the entry to the on-unit is the occurence of a PL/I condition an on-unit for this condition in the bootstrap program will result in a dump being taken before the ERROR on-unit is executed. (For example, if the CONVERSION condition was occurring in the program to be dumped a CONVERSION on-unit could be included in the bootstrap program. Such an on-unit would be entered before the ERROR condition was raised.)
2. Provided that the ERROR on-unit does not include a GOTO out of the on-unit, a FINISH on-unit can be used. Since the standard system action for the ERROR condition is to raise the FINISH condition, the dump will be generated after the ERROR on-unit has been executed.

There is no point in including SUBSCRIPTRANGE or other prefixes in the bootstrap routine, because these are not inherited by called programs.

The bootstrap method is not recommended unless there are particularly strong reasons for avoiding re-compilation.

```
BOOTSTRAP:  PROC OPTIONS (MAIN);

            DCL program* ENTRY EXTERNAL;

            ON ERROR CALL PLIDUMP ('HB',
            'BOOTSTRAP');

            CALL program*;

            END;
*The name of the program to be dumped
should be inserted at the points marked
program* in this example.
```

Figure 12.3. Suggested method of obtaining a dump when re-compilation is particularly undesirable. (See text before using this method.)

CONTENTS OF A PL/I DUMP

The appearance of a typical dump produced by the PLIDUMP modules with the options TFHBA is shown in figure 12.4. The contents of particular sections are described in detail below.

Headings

The dump is headed by the line

PL/I DUMP

```
                                        * * * PL/I DUMP * * *

    USER IDENTIFIER :   EXAMPLE OF PLIDUMP

                                        * * * CALLING TRACE * * *

          (TCA ADDRESS 03D800 )

PLIDUMP WAS CALLED FROM STATEMENT NUMBER 3 AT OFFSET +00009E FROM A ERR  TYPE ON-UNIT WITH ENTRY ADDRESS 03A51C
          (AND DSA ADDRESS 03E050 )

                                                ERROR DIAGNOSTICS

          PL/I CONDITION DETECTED: CONV
          ONCODE    =         612                     SEE LANGUAGE REFERENCE MANUAL
          ONCHAR    =I                                CHARACTER CAUSING CONVERSION ERROR
          ONSOURCE =IF THIS DOES NOT RAISE CONVERSION NOTHING WILL
                                                      STRING CAUSING CONVERSION ERROR

ADDRESS OF ERROR HANDLER'S SAVE AREA 03DE50
           REGISTERS ON ENTRY TO ERROR HANDLER

    REGS 0-7   FF03DE50     0003DE48     0003DDD0     6E03C4C6     0003DBF8     0003DD28     0003DDE0     8003A37A
    REGS 8-15  00000001     0003DD55     00000000     0003B9CE     0003D800     FF03DDF8     4E03C6A8     0003BD52

                                              END OF ERROR DIAGNOSTICS

WHICH WAS CALLED FROM A LIBRARY MODULE WITH ENTRY ADDRESS 03C4C0 (AND DSA ADDRESS 03DDF8 )

WHICH WAS CALLED FROM A LIBRARY MODULE WITH ENTRY ADDRESS 03B770 (AND DSA ADDRESS 03DAE8 )

WHICH WAS CALLED FROM A LIBRARY MODULE WITH ENTRY ADDRESS 03A5D0 (AND DSA ADDRESS 03DD80 )

WHICH WAS CALLED FROM STATEMENT NUMBER 5 AT OFFSET +0000AC FROM PROCEDURE EXAMPLE WITH ENTRY ADDRESS 03A460
          (AND DSA ADDRESS 03DC68 )

                                        * * * END OF CALLING TRACE * * *


          TRACE OF PL/I CONTROL BLOCKS

          TASK COMMUNICATIONS AREA

 ADDR. OFFSET
03D800 00000  00000000 0003DC58 FF03D800 FF048A20  00000000 0003D810 000547B0 0003D9C0  .........Q..........Q.......R.
03D820 00020  00000000 0003DA48 0003D920 D4D9C4C1  0003DA50 00000000 0003DA18 95D9C024  .........R.MRDA.............R..
03D840 00040  0003D9E0 00000000 0003CCAE 00000000  00000000 00000000 80000000 0003B48A  ..R.............................
03D860 00060  0003B48C 0003B638 100A47F0 0003CCA0  0003CCA2 0003CCB2 0003BD52 100C1000  ............0...................
03D880 00080  582E0004 58EE0000 19DF478C 00C29500  C001478C 00BC180E 18E1181F 58FC00AC  .............B..................
03D8A0 000A0  07FF0000 00000000 000007FE 0003B4EA  A4EC58FC 0078051F DB01A166 18DF9834  .....................Y..........
03D8C0 000C0  D0209160 D001078E 9140D001 478C00DC  D203D04C D0509120 D001078E D201D056  ...-..... ......K...........K...
03D8E0 000E0  D0549180 D054071E 181F58FC 00F407FF  17225D20 00000000 0003D8AA 0003D8AA  .............4....).......Q...Q.
03D900 00100  0003D8AA 0003D8AA 00000000 00000000  00000000 00000000 00000000 20004323  ..Q...Q.........................

          TCA IMPLEMENTATION APPENDAGE

 ADDR. OFFSET
03D920 00000  00049000 00000000 0E03BD18 7FFD1030  00000000 00000000 00000000 00000000  ................................
03D940 00020  0003D9F8 0003DA98 0003B5A2 00000000  00000000 00000000 00000000 00000000  ..R8............................


                                        * * * PL/I DUMP * * *

03D960 00040  0003B5F8 0003C30C                                                  ...8..C...........R.......R.....

          LIBRARY WORK SPACE

CONTENTS OF REGISTER SAVE AREA
    REGS 0-7   0003D198     0003A39C     0003E050     4E03B386     0003DBF8     00000000     0003A39C     0003A400
    REGS 8-15  0003E100     0003E120     0003DC68     0003D920     D2008003     FF03DEE8     4E03B3B6     8E03D19C


 ADDR. OFFSET
03DEE8 00000  08000110 0003E050 9C188000 4E03B3B6  8E03D19C 0003D198 0003A39C 0003E050  .............+.....J...J.........
03DF08 00020  4E03B386 0003DBF8 00000000 0003A39C  0003A400 0003E100 0003E120 0003DC68  +.......8.........................
03DF28 00040  0003D920 D2008003 0003DF70 FF03E128  70020080 0003DF60 00440000 00100002  ..R.K.....................-.......
03DF48 00060  F0F0F3F2 F1F245F0 00000000 0000005C  4E049AFC F0C1C35C 40404040 40404040  003212.0.......*+...0AC*
03DF68 00080  40404040 40404040                                                  ........    ............

          DYNAMIC SAVE AREA (ON-UNIT)

CONTENTS OF REGISTER SAVE AREA
    REGS 0-7   FF03E128     0003A39C     5E03A572     0003A308     0003DBF8     00000000     0003DD10     0003A400
    REGS 8-15  0003E100     0003E120     0003DC68     0003D920     7098C030     0003E050     4E03A5BA     0003B380


 ADDR. OFFSET
03E050 00000  8C24E090 FF03DE50 050CB000 4E03A5BA  0003B380 FF03E128 0003A39C 5E03A572  .............+..............;...
03E070 00020  0003A308 0003DBF8 00000000 0003DD10  0003A400 0003E100 0003E120 0003DC68  .......8.........................
03E090 00040  0003D920 7098C030 FF03DEE8 FF03E128  FF03E128 91E091E0 0003DC68 50F09BEA  ..R.........Y.................0...
03E0B0 00060  D2031020 9BBE41F0 000442F0 100858F0  9BAE07FF 58809BE6 D2039BE6 709858F0  K.......0...0...0........WK..W...0
03E0D0 00080  802445E0 96E2D203 0003E090 D2039BD6  70D858F0 802043F0 B00F50F0 B00C54F0  .....SK.....K..O.Q.0...0...0...0
03E0F0 000A0  9BC650F0 9BDA5880 709858F0 802054F0  E3C6C2C8 0003E100 00040000 0000C5E7  .F.0.......0...0TFBH..........EX
03E110 000C0  C1D4D7D3 C540D6C6 40D7D3C9 C4E4D4D7  0003E10E 00120000                    AMPLE OF PLIDUMP..............Y
```

Figure 12.4. An example of PLIDUMP

Abbreviation	Condition Name
AREA	AREA
CHCK	CHECK
COND	CONDITION (programmer named condition)
CONV	CONVERSION
ENDF	ENDFILE
ENDP	ENDPAGE
ERR	ERROR
FIN	FINISH
FOFL	FIXEDOVERFLOW
KEY	KEY
NAME	NAME
OFL	OVERFLOW
REC	RECORD
SIZE	SIZE
STRG	STRINGRANGE
STRZ	STRINGSIZE
SUBG	SUBSCRIPTRANGE
TMIT	TRANSMIT
UFL	UNDERFLOW
UNDF	UNDEFINEDFILE
ZDIV	ZERODIVIDE

Figure 12.5. Abbreviations for condition names used in PLIDUMP trace information.

This is followed by the user identifier, if any, given as the second character string in the argument list of PLIDUMP.

Trace Information

A request for trace information results in the following output:

1. A trace of every procedure, begin block, and on-unit that is active at the time of the call to PLIDUMP. For procedures, the procedure name and statement number from which the procedure was called are given. If the 'H' option is requested, the offset of the statement is given as well as the entry point address and DSA address. Also, if the entry point used is not the main entry point and the statement number option was specified, the main entry name is given.

   For multitasking programs the name of the task variable, its status, and the absolute priority of the task are printed. If no task variable is supplied 'NONE' is printed as the name of the task variable. A dummy task variable will have been supplied see chapter 14.

2. For on-units, the values of any relevant condition built-in functions are given. The type of on-unit is given and, where the cause of entry into the on-unit is not self-explanatory, the cause of entry is also given (e.g., if an ERROR on-unit was entered because of a conversion error, this fact is given in the trace information). The on-unit type is specified, using a three or four letter abbreviation. A list of these abbreviations is given in figure 12.5.

3. When a hexadecimal dump is requested, the entry point address of each active block is also given, together with the address of its associated DSA.

4. When the compiler FLOW option is in effect, the flow statement table is given.

5. If a hexadecimal dump is requested, the address of the TCA is printed at the head of the trace.

6. If either a hexadecimal dump or control block information has been requested, and any ERROR on-units are traced, then the following information is also included:

   a. The address of IBMBERR's DSA.

   b. The contents of the general and floating point registers at the time IBMBERR was called.

   c. If there was an interrupt, the address of the interrupt.

   d. A trace of library DSAs back to the last compiled code DSA.

BYTE 1	PL/I condition if any	BASE NO.	BYTE 1	PL/I condition if any	
X'02'	ZERODIVIDE	320	X'CD'		9250
X'03'	FIXEDOVERFLOW	310	X'CF'		1000
X'04'	SIZE	340			
X'05'	CONVERSION	600	X'D3'		9200
X'06'	OVERFLOW	300	X'D5'		3500
X'07'	UNDERFLOW	330	X'D7'		4050
X'08'	STRINGSIZE	150	X'D9'		5050
X'09'	STRINGRANGE	350			
X'0A'	SUBSCRIPTRANGE	520			
X'0B'	AREA	360	X'DF'		5000
X'0C'	ERROR	009	X'E1'		9050
X'0D'	FINISH	004	X'E3'		1000
X'0E'	CHECK	510	X'E5'		4000
X'0F'	CONDITION	500	X'E7'		xxxx
X'10'	KEY	050	X'E9'		4050
X'11'	RECORD	020	X'EB'		0003
X'12'	UNDEFINEDFILE	080	X'ED'		1000
X'13'	ENDFILE	070	X'EF'		1550
X'14'	TRANSMIT	040	X'F1'		1500
X'15'	NAME	010	X'F3'		2000
X'16'	ENDPAGE	090	X'F5'		3768
X'17'		-	X'F7'		3000
X'18'		-	X'F9'		3800
			X'FB'		3900
			X'FD'		9000
			X'FF'		8090

Note: Meanings are only given where there is a directly associated PL/I condition.

Figure 12.6. Error code field lookup table

File Information

A request for file information results in the following output:

1. The default and declared attributes of all open files are given.
2. Buffer contents of all buffers are given. If a hexadecimal dump has been requested, the contents of the buffers are given in both hexadecimal and character notation. If no hexadecimal dump is requested, the contents are given in character notation only.
3. The contents of the FCBs, DCBs,

DCLCBs, IOCBs, and exclusive file blocks are given in formatted hexadecimal notation, if either the 'H' or 'B' option is also included.

Hexadecimal Dump

The hexadecimal dump is produced by the execution of a SNAP macro instruction. Thus the normal SNAP dump is produced. This is fully described in the Programmers Guide to Debugging.

It should be noted that the PSW will contain the address of an instruction in IBMBKMR, one of the modules used to implement PLIDUMP. This will bear no relation to the error in the dumped program.

If the program is not multitasking the SNAP macro specifies all register save areas, subpools, task control blocks, and, provided the O (Only) option is not included in the PLIDUMP options, the trace table.

For a dump of a multitasking program the contents are:

In the control task

- Register save areas
- Subpools
- Trace table
- Control blocks

In the other tasks

- Register contents
- Register save areas
- Subpools
- Jobpack Area
- Linkpack area

Block Option

When the block option is used, the contents of the TCA, the TIA (TCA appendage), and the DSAs in the LIFO stack (that is, all active DSAs) are printed in hexadecimal and character format. The absolute address is printed in the left hand column; the offsets within the block are then printed. This is followed by the contents of the block, first in hexadecimal and then in character notation. For DSAs, the type of DSA is shown; i.e., library DSA, procedure DSA, on-unit DSA, or dummy DSA. The contents of the FCBs, DCLCBs and IOCBs for any open files are printed in a similar format.

In a dump of a multitasking program the contents of the tasking appendage is also printed.

If the option A(all) is used in a multitasking programming the TCA, TIA, DSAs and tasking appendage of all directly ascending tasks will be printed. FCBs, IOCBs, DCLCBs will be printed after files open in any task if the option A is used.

Section 2: Recommended Debugging Procedures

The main difficulty in reading a dump of a PL/I program is knowing where to start. The signposts known to assembler language programmers are of little help. There are, however, five main sources of information to be considered when using a dump to debug a PL/I program. They are:

1. The statement number and the address where the error occurred (if the dump was taken after an error)
2. The type of error (if the dump was taken after an error)
3. The values in the general registers when the dump was taken or when the error occurred
4. The chain of DSAs
5. The TCA

The first two of these items hold equivalent information to that held in the PSW in an OS system dump. The last three items enable the housekeeping to be checked and the location of the control blocks and the program variables to be discovered. The methods of locating other information, given in section 3, refer to the key areas shown above.

Note: Meanings are only given where there is a directly associated PL/I condition.

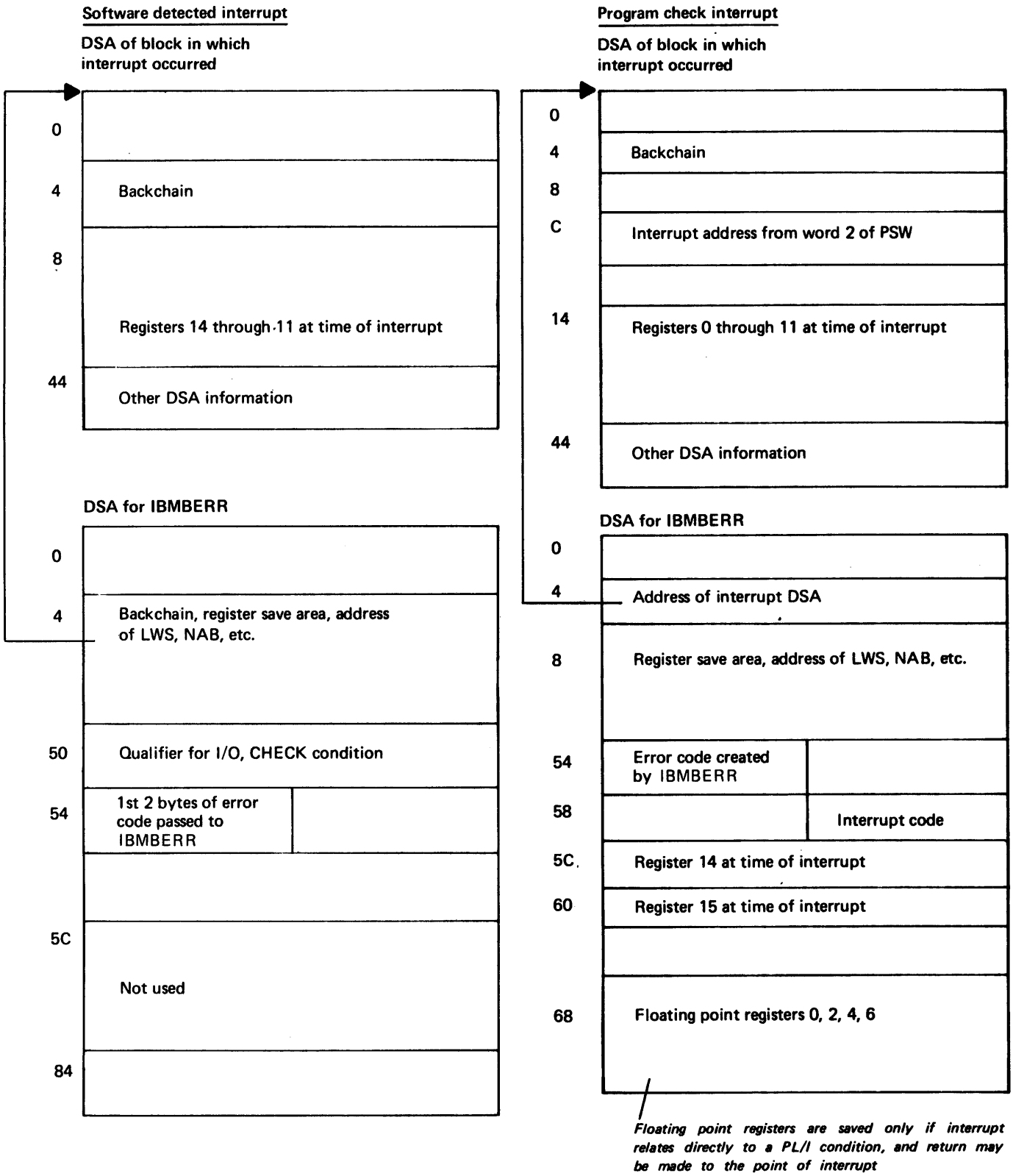


Figure 12.7. The contents of IBMBERR's DSA after a system detected and a PL/I interrupt

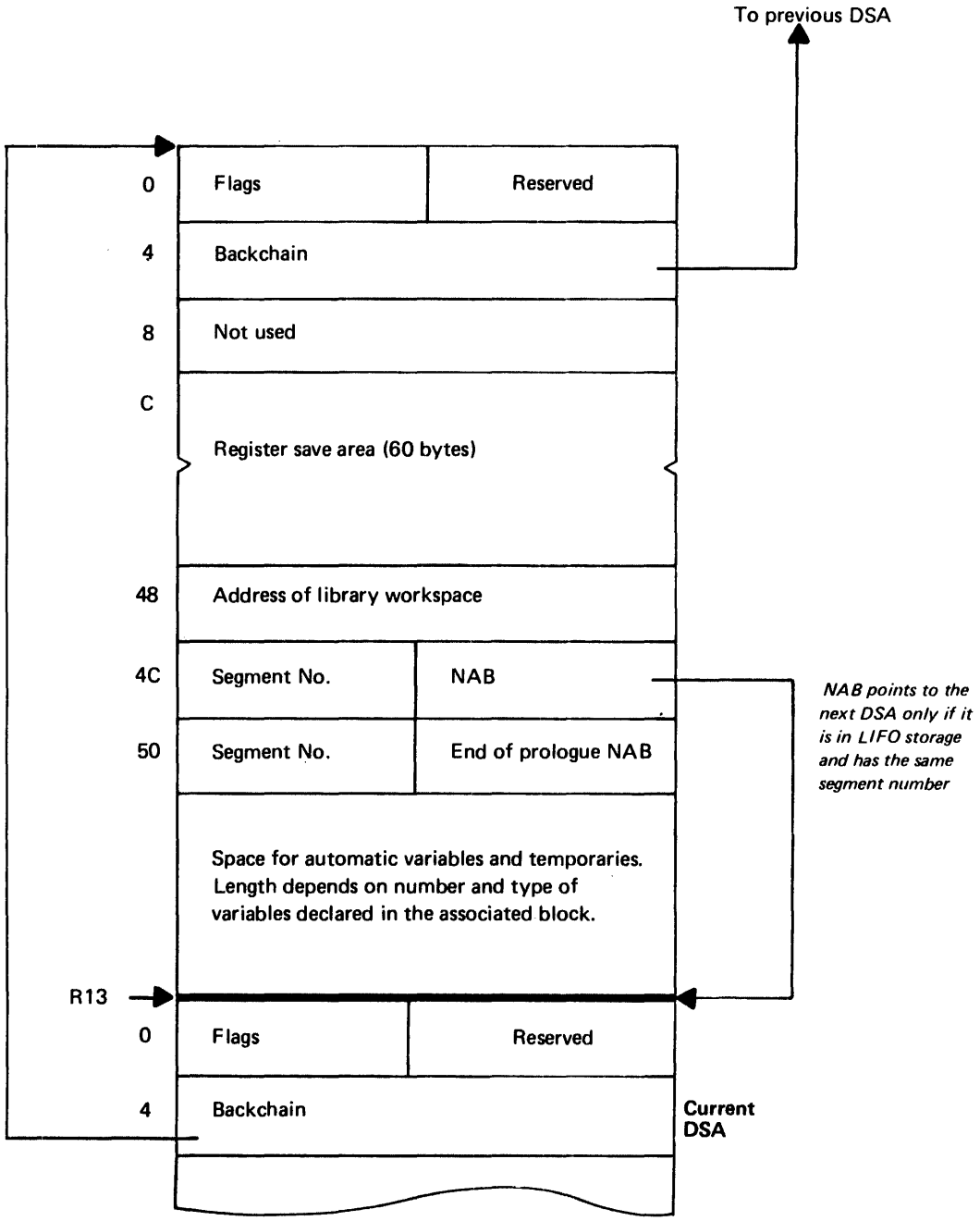


Figure 12.8. The chaining of DSAs

When debugging, it is essential to have a listing of the object program, a variables offset map and a linkage editor map. The object program listing allows the programmer to study the instructions that are being carried out and to find various control blocks in static storage. The linkage editor map allows the programmer to identify particular parts of the executable program phase and, to identify the routine associated with each DSA.

Note: The PSW in the SNAP dump should not be consulted. This will give the address at which the SNAP macro instruction was issued. This is an address in one of the PLIDUMP modules and is not relevant to the error in the problem program.

DEBUGGING PROCEDURES

The best approach to a dump depends on the problem to be solved and must therefore be left largely in the hands of the programmer. However, two suggested courses of action are given in this section.

These courses cover two situations:

1. When PLIDUMP has been called from an ERROR or other on-unit
2. When only an OS ABEND dump has been generated.

Other possible situations are when a dump is taken at a specified point in the program, or when a stand-alone dump is taken. No attempt is made to suggest a course of action in these circumstances. However, in such cases, the main storage situation can be investigated by following the methods itemized in section 3 of this chapter.

Throughout each of the two recommended procedures given in the following paragraphs, there are cross-references to the methods given in section 3. The cross-references consist of the keys by which the methods are identified; for example, H6, D5.

PL/I Dump Called from On-Unit

If a PL/I dump is called from an ERROR on-unit it can be assumed that the housekeeping system of the program is working. If it were not working, the dump would probably not have been generated.

A large amount of diagnostic information will be available at the head of the dump. An error message will have been generated, and this will provide a useful starting point. The first step should be to examine the type of error and the point at which it occurred. ONCODE and other condition built-in function values should be examined, as should the trace information. A suggested procedure is the following:

1. Examine the error by means of the ONCODE and any other relevant built-in function values. These values are given in the trace information. (The meanings of oncodes are given in the Language Reference manual for this compiler.)
2. Find the location of error (P1) and in which block the error occurred (H12). If error occurred in library module, see H14. This information is normally available from the head of the PLIDUMP.
3. Examine the trace to see if it appears as expected.
4. Examine the information in the file buffers, and check that file attributes are as expected. This information will be printed in the dump heading.
5. Check the values of any variables involved in the interrupt (V1-V6).
6. Check values of registers to see if dedicated registers are pointing to correct areas (H8 & H9). Distinguish between compiled code and library register usage.
7. If SUBSCRIPTRANGE or STRINGRANGE is not enabled, check that the error was not caused by one of these conditions.
8. Check housekeeping (H1-H16) starting with area most directly concerned with type of statement in which the error occurred.
9. Check values of all variables in the program (V1-V6).
10. Check logic of code being executed from object listing.

OS ABEND Dump

Provided a SYSABEND or a SYSUDUMP card is included in the JCL an OS ABEND dump will be generated when there is a failure of the error-handling modules, or of the module that prints the PL/I hexadecimal dump. It

should be noted that the failure of these modules is more likely to be caused by the overwriting of essential information than by an error in the modules themselves.

Because ABENDS caused by overrunning the specified time (SYSTEM 322) do not enter the STAE exit, these will cause dumps to be generated in normal circumstances.

An ABEND dump will not normally be produced for program checks, because a program check exit is set by the PL/I housekeeping routines, so that the system returns all program checks to the error handler. In the error handler itself, the program check exit is reset so that a program check interrupt will result in a dump.

Thus, an ABEND will be produced if the program interrupt exit, which is normally set by the program initialization routines to prevent a dump, has been reset during the program, or, possibly, has not been set at all. The second alternative is extremely unlikely. A third possibility is that the program check exit itself is not working, and the SPIE macro in the initialization routines did not successfully set the program check exit. The most probable of these suggested causes is that the program check exit has been reset by the program. The program interrupt exit is always reset for the duration of error handling or PLIDUMP, to prevent looping should an interrupt occur. (See chapter 7, "Error Handling.") If an interrupt occurs during error handling, an ABEND with a code of 4000 is produced. This will result in a dump if SYSABEND or SYSUDUMP cards have been provided. An interrupt in the error-handling routines indicates either that the error-handling routines are at fault, or, more probably, that some of the control information of the error-handling routines has been overwritten during the execution of the program. The most practical solution may be to re-run the program with SUBSCRIPTRANGE, STRINGSIZE, and STRINGRANGE enabled.

These PL/I conditions check for possible overwriting by subscripts or substrings that are beyond the bounds of the variable referred to.

However, having obtained an ABEND dump, the following debugging procedure may be adopted.

1. Determine whether the dump was caused by an interrupt in the error handling routines or a housekeeping error discovered during the analysis of an ABEND. If the cause was an interrupt in the error handler a message will have been sent to the console before the ABEND was issued, and the ABEND will have a code of 4000, if the interrupt occurred in one of the error handling routines. Note that codes 322 and 122 may also give system dumps. And that the use of NOSPIE or NOSTAE can result in the generation of a dump.

2. Locate instruction causing interrupt. This is done by looking for the PSW (01).

3. Inspect this instruction to see if it appears to have been overwritten, bearing in mind the cause of the interrupt, e.g.,

   a. do the registers used in the instruction contain incorrect information, picked up because of overwriting?

   b. is it a branch to a protected address?

4. Inspect the TCA(05) to ensure that all error-handling addresses are correct.

5. Investigate the housekeeping fields, starting with the DSA chain (H1-H3), then the chain of ONCAs (H5,H6).

6. Investigate the error that caused entry into the error handler. This can be done by examining the contents of IBMBERR's DSA (H7) and the associated ONCA (H6). See whether incorrect information passed to the error handler could be causing a failure.

7. Check for uninitialized variables (particularly pointers), and incorrect passing of parameters.

8. If none of the above produces a solution, an error in the error-handling modules is a possibility. If you decide to call IBM for assistance at this point, refer to appendix C in the Programmer's Guide for this compiler. The cause of the original entry to the error handler may have been discovered, and can, perhaps, be avoided by altering the source program so that the error does not occur. It must be emphasized that the cause of entry into the PL/I error handler was <u>not</u> the cause of the system dump.

9. If the interrupt is not in the error handler, or one of the routines it calls, the highest probability is still that the program check exit was altered in the error handler and that an invalid branch has been made from one of the addresses in the TCA

because of overwriting. A careful check should therefore be made in the TCA. (See appendix A for map of TCA.) If this fails to produce results, return to stage 2 of the above procedure.

Section 3: Locating Specific Information

This section tells the reader how to discover information from the dump. The section has been produced in a modular form for easy reference. The reader should look through the contents list below to discover the items in which he is interested. Suggested methods of debugging a PL/I program from a dump are given in section 2 of this chapter. Unless the programmer is experienced in using dumps, or is looking for some particular item, the procedures in section 2 should be followed, rather than attempting to find various items through the information in this section.

CONTENTS

Key Areas of a PL/I Dump

P1 Statement number and address where error occurred (dump called from on-unit only)

P2 Type of error (dump called from on-unit only)

P3 Register contents at time of error or dump invocation

P4 The DSA chain

P5 The TCA

Key Areas of an ABEND Dump

O1 Finding address of interrupt
O2 Type of interrupt
O3 Register contents at point of interrupt
O4 The DSA chain
O5 The TCA
O6 Finding the program interrupt element (PIE)

Stand-alone Dumps

S1 Finding key areas in stand-alone dumps

Housekeeping Information in all Dumps

H1 Following the DSA backchain

H2 Associating instruction with correct module

H3 Following calling trace

H4 Associating DSA with block

H5 Finding relevant ONCA

H6 Following the chain of ONCAs

H7 Finding information from IBMBERR's DSA

H8 Finding and interpreting register save areas

H9 Register usage

H10 Following free-area chain

H11 Finding the task variable

H12 Block structure of program (static-backchain)

H13 Forward chain in DSA's

H14 Action if error is in a library module

H15 Discovering contents of parameter lists

H16 Finding main procedure DSA

H17 Finding the relationships between tasks

H18 Finding the tasking appendage

H19 Finding the TCA from the tasking appendage

Finding Variables

V1 Automatic variables

V2 Static variables

V3 Controlled variables

V4 Based variables

V5 Area variables

V6 Variables in areas

Control Blocks and Fields

C1 Quick guide to identifying control fields

KEY AREAS OF A PL/I DUMP

P1: Statement Number and Address where Error Occurred (Dump Called from On-Unit only)

Information required is the point at which the condition that caused entry to the on-unit occurred. This is identified in the trace information. If no trace information is generated, the method suggested for ABEND dumps can be employed. If the condition occurred in compiled code, the machine instruction being executed can be identified on the object program listing. This is done by subtracting the address of the program control section from the address of the interrupt and looking at this offset in the object program listing. The instruction thus found will be the one after the instruction that was last executed.

Note: If PLIDUMP is called a number of times in a program a different user identifier should be used with each CALL statement so that the point at which the dump was taken is obvious.

P2: Type of Error (Applies to Dump Called from On-Unit only)

The type of error is identified in the trace information, in terms of the type of on-unit entered and the reason for entry. The ONCODE is also given, thus providing further indication of the cause of the condition. If the dump was called from an ERROR on-unit, an error message should have been generated before the dump. This again will give the cause of the error.

If no trace information has been generated, the type of error can be discovered from the error code appearing in the ONCA associated with the interrupt. The method for finding the ONCA is described in H5.

P3: Register Contents at Time of Error or Dump Invocation

If trace information has not been generated, the contents of the registers can be found from the save area in the DSA. The addresses of all DSAs appear in the trace information. The register contents required will depend on the situation. If PLIDUMP was called from an on-unit, the register contents at the time the condition was raised will be most useful, unless the condition was raised in a library module. If the condition was raised in a library module, the contents of the registers at the point where the library call was made will probably prove more useful.

For a dump called from an on-unit the method of finding the register contents is as follows:

1. Find the DSA of IBMBERR. The value of register 13 will be found in the chainback field at offset 4 of this DSA. The first byte will contain the segment no.(probably 'FF') and can be ignored for addressing purposes.

2. If the interrupt was a program check interrupt (see figure 12.7), the contents of registers 14 and 15 will also be stored in the DSA, register 14 at offset '5C'(92) and register 15 at offset '60'(96) from the head of the DSA.

3. Registers 0 through 11 will be stored in the save area of the previous DSA, starting at offset '14'(20).

4. If the interrupt was a software interrupt, the registers will be stored at offset 'C'(12) of the DSA before IBMBERR's DSA in the order 14 through 11. See figure 12.7.

Discovering if interrupt was program check interrupt: If trace information is available, a check can be made on whether IBMBERRA or IBMBERRB was called. IBMBERRA is entered after program check interupts, IBMBERRB after software interrupts. If no trace information is available, the simplest method of discovering if the interrupt was a program check interrupt is to inspect bit 7 in byte X'56' (86) in IBMBERR's DSA. This is set to zero for program check interrupts, and to 1 for other interrupts.

Finding register values if interrupt occurred in library routine: If on-unit

was entered from a library module, a search back through the DSA chain to the first compiled code DSA should be made. This can be discovered from the trace information or by following the backchain from IBMBERR's DSA (offset 4 in each DSA) until a procedure block, begin block, or on-unit DSA is found. This may be determined from flag bits 4 and 5 of a DSA, as follows:

Bit 4	Bit 5	DSA
0	0	Procedure block
1	0	Begin block
1	1	On-unit

The value of register 12 can only be discovered in a DSA prior to a compiled code DSA, as it is not stored by library routines when they are entered. This means that the dummy DSA always contains the value of register 12. Register 12 should point to the TCA, whose address is also given at the head of trace information.

No trace information generated: If no trace information has been generated, the register values on taking the dump will be printed at its head. The address of the DSA for PLIDUMP will be in register 13. The chainback can then be followed to find the DSA for IBMBERR. The DSA for IBMBERR can be recognized if an on-unit is involved, because it will be the DSA before the on-unit DSA. IBMBERR's DSA will always be headed by a flag byte of hexadecimal '88' meaning that it is a library DSA in LIFO storage. To identify IBMBERR's DSA for certain, register 15 of the previous block's DSA must be inspected to see if it points to the module IBMBERR.

P4: The DSA Chain

The addresses of the DSAs are given in a PL/I dump if trace information and a hexadecimal dump are requested. If trace information is not requested, the address of the DSA for the dump routine can be obtained from register 13 at the head of the dump. The chainback field is held in the second word of the DSA. When the dummy DSA is reached, this chainback field will be set to zero. The DSA chain passes through DSAs in LIFO storage and DSAs in LWS (library workspace).

See H1 and figure 12.9 for details of how to follow the DSA chain.

P5: The TCA

The address of the TCA is given in a PL/I dump. If 'B' (block option) is specified in the dump-options character string, the complete TCA (including the appendage) is printed separately from the body of the dump.

The TCA is addressed by register 12. The format of the TCA is given in appendix A. The use of the various fields is explained in chapter 4.

KEY AREAS OF AN ABEND DUMP

01: Address of Interrupt

If the ABEND code is 4000 the address of the interrupt can be found from the second word of the PSW, which gives the address of the instruction following the point of interrupt. The PSW is held in subpool 5. A description of how to find the PSW is given in the publication OS: Programmer's Guide to Debugging. The associated statement number in the source program can normally be found by finding the last compiled code DSA, and finding the point at which the exit was made (register 14 in the save area). The address of the program control section can then be subtracted from this address, and the offset compared to the listing will give the appropriate statement number.

Finding the statement number is not likely to prove useful because of the circumstances in which an OS system dump is generated. The address found will usually be the address at which the error handler was entered before the program check exit was altered. The reason for entry into the error handler is not the cause of the dump. If the ABEND code is not 16000 see 06.

02: Type of Interrupt

The type of interrupt can be found from the first word of the PSW (see Principles of Operation for details).

03: Register Contents at the Point of Interrupt

Registers 14 through 2 appear in the PIE (program interrupt element). Registers 3

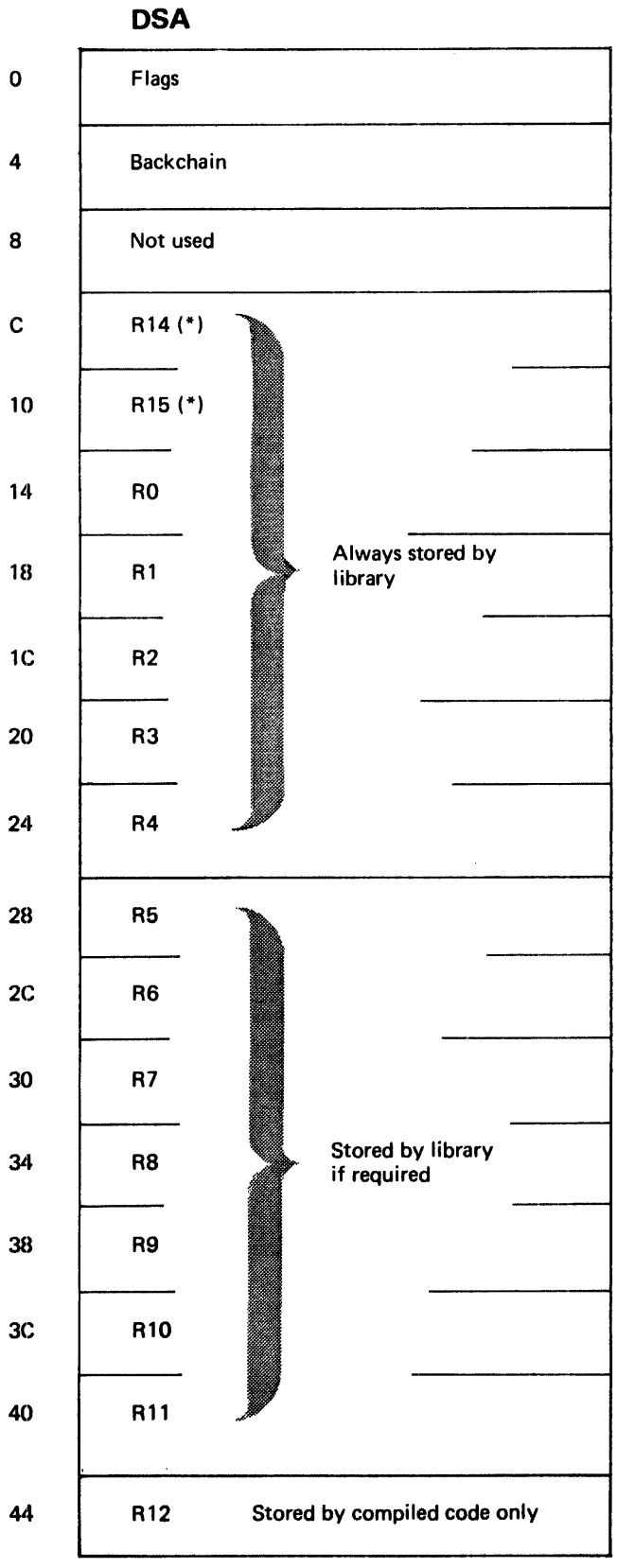


Figure 12.9. The register save area in the DSA

through 13 are those printed in the save area trace. See O6 for finding the PIE.

O4: The DSA Chain

Register 13 should point at the most recent DSA. The back chain can be followed from offset '4' of each DSA. See figure 12.9.

O5: The TCA

Register 12 should point at the TCA.

O6 Finding the Program Interrupt Element (PIE)

The program interrupt element (PIE) will be found at the head of subpool 5. The PIE will be followed by registers 3 through 13 and then the STAE work area. The STAE work area holds the last problem program PSW. This is the value required for finding the original cause of the ABEND if the ABEND code is other than 4000.

STAND-ALONE DUMPS

S1: Finding Key Areas in Stand-alone Dumps

The programmer should attempt to find the various PL/I key areas (TCA, DSA chain, etc.) discussed above.

Further information on reading stand alone dumps is given in the publication OS: Programmers Guide to Debugging.

HOUSEKEEPING INFORMATION IN ALL DUMPS

H1: Following the DSA Backchain

Each DSA holds a backchain address in the second word. This word holds the address of the previous DSA. The end of the chain is marked by the dummy DSA whose first word contains the flag hexadecimal '82'. The backchain in the dummy DSA points to the external save area or is zero if the program was called from the system. (See P4 or D4 for finding the DSA chain).

For programs using multitasking the DSA backchain leads to the dummy DSA of the major task. The DSA of the block in which the task was attached is not included in the chain. To find this DSA the 'static' backchain held at offset X'58' (88) can be used provided the procedure attached as a task is internal to the attaching block. If the procedure is not internal the NAB value X'4C' (76) in the DSA before it will normally point to the required DSA.

(The method of chaining during a multitasking program is explained in chapter 14. For relationship of NAB and DSA chaining see H13.)

H2: Associating Instruction with Correct Statement and Program Block

Statement Number and Program Block: The statement number and entry point associated with the interrupt will normally be given in a PLIDUMP. However, if they have to be found by the programmer, he should follow the method used by the error message modules.

Statement number: It must first be established whether the GOSTMT option is in effect. This will be indicated in the listing for the compilation. If the listing is not available it will be flagged in the compiled code DSA. (Flag bit 13 of the DSA flags is set to '1'B.) If this bit is not set the table of offsets and statement numbers may be available, if this is not available statement numbers and offsets must be deduced from the object program listing. The method of using the table of offsets is described below under the heading "Using the Table of Offsets". If both statement numbers and the table of offsets are available it will probably be faster to use the table of offsets rather than the statement number table.

The statement number is found by use of the DSA chain as described below:

1. Find the chain of DSAs. The most recent DSA should be addressed by register 13.

2. If the DSA found is not a compiled code DSA, (in a compiled code DSA flag bits 4 and 5 are set to '00'B, '01'B or '11'B) the interrupt was not in compiled code. If the interrupt was in compiled code, the interrupt address can be directly associated with a statement number.

   If the interrupt was not in compiled code, the address at which compiled

code was left must be discovered and this address associated with a statement number. To find the address at which compiled code was left:

a. Chain back along the DSA chain until a compiled code DSA is reached (flag bits 4 and 5 set to '00', '01', or '11'B).

b. The register 14 address saved in the DSA (offset 12X'C') will be the point to which the library module or other module would have returned if the call had been successfully completed.

The address thus found is the address to be associated with a statement number.

3. Chain back one DSA to the DSA before the compiled code DSA that has been discovered in 1 or 2 above. The register 15 value in this DSA (offset 16 X'10') is the entry point of the block. If this appears to give an invalid result, check to see whether the DSA is one of those used in interlanguage communication (flag bit 7 set to '1'B and bit 0 of flags 2 (offset X'76') set to '1'B). If this is the case chain back one more DSA and try again.

4. At offset 8 from the entry point of the block, the address of the statement number table will be held.

5. Calculate the offset between the value in the first word of the statement number table and the address for which a statement number is required. If the address for which a statement number is required is less than the address in the first word of the statement number table, then either an invalid branch has been made, or a compiler generated subroutine is being executed. If it is possible that a compiler generated subroutine is being executed return to the compiled code DSA and attempt to find a statement number associated with the values held first in register 6, and, if this gives an invalid or improbable result, then in register 14. If the second word in the statement number table is less than the offset between the address for which a statement number is required and the first word of the statement number table, it is not within the program control section and an erroneous branch has been made out of the program.

6. If the offset is more than X'7FFF' the statement number will be held in the second or subsequent sections of the table. Obtain the number given by translating the offset into binary and ignoring the last 15 bits and step down this number of sections of the table. (For example, if the offset was X'8FFF', translate to binary = '1000 1111 1111 1111'B, ignore last 15 binary digits =1, therefore step down one section of the table. If the offset was X'18FFF' the binary would be '0001 1000 1111 1111 1111'B. Ignoring the 15 right hand bits leaves '11'B therefore step down three sections of the table.)

   The address of the second section of the table is held at offset X'8' in the table, the address of the third section is held at the head of the second section, the address of the fourth section at the head of the second section and so forth.

7. When the correct section of the table has been identified, search for the first offset in the table that is greater than or equal to the offset that is being searched for. Following this offset the statement number is given in two-byte hexadecimal format.

Procedure name: To find the entry point name, a chainback is made beyond the first procedure DSA found on the chain. Register 15 in the save area before this procedure DSA will point to the entry point of the procedure. (Procedure DSA have flag bits 4 and 5 set to '00'B. The register 15 value is held at offset 16 X'10'.)

The entry is preceded by a one byte field that holds the number of characters in the name. This one byte field is in turn preceded by the entry point name.

Using the table of offsets: Statement numbers can also be found by comparing them with the offsets in the offset and statement number table generated by the compiler when the OFFSET option is specified.

Offsets are held from each primary entry point or a procedure or on-unit. To use the table of offsets find the entry point used by the program in the manner described above. Find the primary entry point for the procedure. (If the primary entry point was not used look at the object program listing to see the relationship between the entry point used and the primary entry point.) Note, the offsets given are from the point marked *REAL ENTRY in the object program listing. This point is one byte after the end of the primary entry point name.

If the interrupt occurred in an on-unit

it may be necessary to discover the type of on-unit entered before it can be identified. This is done by inspecting the DSA before the DSA of the on-unit. This DSA will be for IBMBERR. At offset 84 (X'54') in this DSA the first byte of the error code will be held. Compare this with the values in figure 12.8. This will given an associated PL/I condition. It will be the on-unit for this condition that has been entered. If there is more than one on-unit for the condition, the on-unit entered must be deduced by studying the dump, and source and object listings. If the register 15 value appears to be invalid this may be caused by rechaining in interlanguage processing (see chapter 13). If this is possible, chain back one more DSA and try again. (To check if this has occurred see 3, above under "Statement Numbers").

H3: Following Calling Trace

The calling trace can be followed because branches within the program are always made on registers 14 and 15. Hence register 15 in each DSA points to the address that was branched to from that block. Register 14 points to the address to which control passed when the block was completed. By finding the entry point name (see H2 above) it is possible to follow the calling trace.

H4: Associating DSA with Block

DSAs are associated with code by finding the register values in the preceding DSA register save area (H8) and using the fact that all branches are made via registers 14 and 15. Register 14 in any DSA points to the instruction after the point at which control left that block. Register 15 points to the address at which the next block was entered. The block in the source program can be identified by statement numbers or entry point, found as described in H2, above.

H5: Finding Relevant ONCA

When an interrupt has occurred in the error handler and a system dump has been produced, it is possible to discover the information that the error handler would have used to generate appropriate error messages. The ONCA holds values for the condition built-in functions. The appropriate ONCA can be found in the following manner.

1. Find the DSA before that of IBMBERR (follow back the DSA chain until register 15 in the save area points to IBMBERR). See H1, H3, H7. If this is a library DSA (flag bits 4 & 5 set to '10') go to 3, below.
2. Find the LWS addressed from this DSA. The address is held at offset X'48' (72).
3. Find the offset from the LWS to the ONCA. This is held at offset 2 in the LWS.
4. Add the offset to the address of the library DSA in LWS.

H6: Following the Chain of ONCAs

ONCAs are used to hold condition built-in function values. They are chained together, one being provided for every level of interrupt. The chainback field is in the first word of the ONCA. The dummy ONCA is marked by a chainback field of zero.

H7: Finding Information from IBMBERR's DSA

The information held in IBMBERR's DSA is used by the error message modules for information about the error. If the messages have not been generated the information can be deduced from the DSA. The contents of IBMBERR's DSA are shown in figures 12.7. See H4 for associating DSAs with correct code.

H8: Finding and Interpreting Register Save Areas

Register save areas are held at offset X'C'(12) in all DSAs, including DSAs in LWS. Offsets and registers are shown in figure 12.10. Each DSA holds the register values as they were on exit from its block.

Note: Library routines store at least registers 14 through 4, and up to registers 14 through 11; compiled code routines store registers 14 through 12. Thus the address of register 12 can always be found in the dummy DSA although it may not be in other DSAs. The contents of the register save area in the DSA of the block that called

Register	Compiled code usage	Library usage
R0	Work register	Work register
R1	Work register	Work register
R2	Program base (*,**)	Work register
R3	Static base (**)	Program base (**)
R4	Work register	Work register
R5	Work register	Work register (if used)
R6	Work register	Work register (if used)
R7	Work register	Work register (if used)
R8	Work register	Work register (if used)
R9	Work register	Work register (if used)
R10	Work register	Work register (if used)
R11	Work register	Work register (if used)
R12	TCA pointer (**)	TCA pointer (**)
R13	Current DSA pointer (**)	Current DSA pointer (**)
R14	Branch register	Branch register
R15	Link register	Link register

(*) The contents of the program base register are saved during in-line record I/O and TRT instructions

(**) Dedicated register, i.e., the contents remain unchanged throughout the execution of the associated compiled code or library routine

Figure 12.10. Register usage

IBMBERR are slightly different from normal if the interrupt was a hardware interrupt. See figure 12.7.

H9: Register Usage

Register usage is fully discussed in chapter 2, "Compiler Output." A summary of register usage, showing which registers are always used for a particular purpose, is given in figure 12.10.

H10: Following Free-Area Chain

The free-area chain connects the areas of non-LIFO dynamic storage that have been used and freed, but have not been absorbed into the major free area. See chapter 6, "Storage Management." The chain starts at offset 8 in the implementation-defined appendage, which is addressed from offset X'28'(40) in the TCA. The end of the chain is marked with a zero entry.

H11: Finding the Task Variable

The task variable is held in the TCA at offset X'24' (36).

H12: Block Structure of Program (Static Backchain)

The block structure of the program can be followed from the address held at offset X'58'(88) in each compiled code DSA. This address holds the address of the compiled code DSA of the statically encompassing block. The chain thus formed is known as the static backchain.

H13: Forward Chain in DSAs

The forward chain in DSAs is not supported by the compiler. However, a forward chain through the LIFO stack can normally be followed by use of the NAB pointer. The NAB pointer is held at offset X'4C'(76) from the head of each DSA. The last pointer in the chain points to the major free area. If the NAB pointer contains anything except 'FF' in its first byte, the chain cannot be followed, because it is not contained in a single LIFO segment. The address required is held in the last three bytes of NAB; the first byte contains the segment number (see C1). The forward chain includes only those DSAs in the LIFO stack and does not include any DSAs in LWS.

H14: Action if Error is in a Library Module

The fact that the interrupt or the error was discovered during the execution of a library module suggests that a check must be made on the data that is being passed to the module.

To discover the contents of a parameter list see H15.

H15: Discovering Contents of Parameter Lists

Parameters are passed in a list of words pointed to by register 1, except during stream I/O. To find the position of a parameter passed to a program, find the value of register 1 in the save area of the DSA (see H4) of the calling block. Register 1 will then locate the parameter list. If the list is in static storage, this can be compared with the static storage listing. The name of the called routine can be discovered (H3). The correct parameters for PL/I library routines are given in the appropriate library PLM.

H16: Finding Main Procedure DSA

The main procedure DSA can be found by following the backchain of DSAs to the dummy DSA. The address of the main procedure DSA will be given by the last 3 bytes of NAB in the dummy DSA. NAB is held at offset X'4C'(76) in the dummy DSA. The address of the dummy DSA is held at offset X'24'(36) in the TCA appendage, which is addressed from offset X'28'(40) in the TCA. The dummy DSA can be recognized by the presence of X'82' in the flag byte.

H17: Finding the Relationship between Tasks

The relationship between tasks can be discovered from the chains in the tasking appendage. The chain held at offset X'28' (40) points to the tasking appendage of the most recently attached subtask.

The chain at offset X'24' (36) points to the task with the same attaching task that was attached before the task being inspected (elder sister). If there is no such task the field is set to zero.

The chain at offset X'20' (32) points to the subsequently attached task with the same attaching task (younger sister). If there is no younger sister this chain points to an offset within the tasking appendage of the parent task. An attempt to continue along the chain results in a zero field being met. (See figure 14.7.)

To Find the Parent Task

Search along the chain held at offset X'20' (32) in each tasking appendage. When this field is zero the tasking appendage of the parent task has been reached. The start of this tasking appendage is at an offset of X'-8'(-8) from the address held in the pointer of the previous tasking appendage. (See figure 14.7.)

To Find all Subtasks of a Task

The address of the most recently attached subtask is held at offset X'28' (40) in the tasking appendage. Other subtasks can be found by following the chain held at offset X'24' (36) in the tasking appendage until a zero field is reached. This will be the end of the chain and is the first of the active subtasks to be attached by the task. (See figure 14.7.)

To Find Sister Tasks

Previously attached sister tasks (elder sisters) can be found by following the chain held at offset X'24' (36) in the tasking appendage.

Subsequently attached sister tasks (younger sisters) can be found by following the chain held at offset X'20' (32) in the tasking appendage. When a zero field in this chain is reached, the parent task has been found. The most recently attached sister task is the last one whose chain field does not hold a zero value. The word after the zero value will point to the tasking appendage of this task.

The method used for chaining tasks is explained in chapter 14, and shown in figure 14.7.

H18: Finding the Tasking Appendage

The address of the tasking appendage is held at offset X'2C' (44) in the TCA and at offset X'50' (80) in the dummy DSA of the attaching task.

H19: Finding the TCA from the Tasking Appendage

The TCA is addressed from X'2C' (44) in the TCA tasking appendage.

FINDING VARIABLES

The value of the variables in the program at the point of interrupt can be discovered by using the compiled code listing as a guide to their addresses, and then finding these addresses in the dump. The method used depends on the type of variable.

V1: Automatic Variables

Automatic variables can be found by using an offset from the DSA of the block in which they were declared. This information appears in the variables offset map generated when the compiler MAP option is used. If the compiler MAP option has not been used the information can be deduced from compiled code. (For finding DSA associated with block see H4).

V2: Static Variables

Static variables are normally addressed by an offset from register 3. This offset is given in the variables offset map generated when the compiler MAP option is used. If the compiler MAP option has not been used the offset can be deduced by studying the listing of compiled code. The value of register 3 can be found in the save area of the DSA. (For finding DSA associated with block see H4).

V3: Controlled Variables

As described in chapter 2, controlled variables are addressed by an anchor word that is held in the pseudo-register vector. This can be identified from compiled code.

The address in the pseudo-register vector is the address of the data or, in certain circumstances (see appendix A), of a descriptor or a locator/descriptor. The data is preceded by a control block - the controlled variable control block. The address of the previous allocation is held at an offset of -8 from the address in the PRV. If there is no previous allocation, the address is set to zero.

V4: Based Variables

Based variables are located by finding the value of the defining pointer. This value is found by using one of the methods described above to find static, automatic, or controlled variables. If the pointer is itself based, its defining pointer must be found and the chain followed until the correct value is found.

Typical code would be the following:

For X BASED (P), with P AUTOMATIC

```
58 60 D 088          L 6,P

58 E0 6 000          L 14,X
```

P is held at offset X'88' from register 13, and this address points at X.

Care must be taken when examining a based variable to ensure that the pointers are still valid.

V5: Area Variables

Area variables are located in one of the ways described above, according to their storage class.

Typical code would be:

For area variable A declared AUTOMATIC

```
41 60 D 088            LA 6,A
```

The area would start at offset X'88' from register 13.

V6: Variables in Areas

Variables in areas are found by locating the area and then using the offset to find the variable.

CONTROL BLOCKS AND FIELDS

For simplicity, the methods of finding various control blocks are placed in an alphabetic table. Details of the control blocks can be discovered from the relevant

chapters (see index) or from appendix A.

As well as control blocks, various other items are included in the list. Where necessary, cross-reference is made to other sections in this chapter.

C1: Quick Guide to Identifying Control Fields

Automatic variables	see "Variables"
Backchain	
DSA backchain	offset X'4' in DSA
ONCA backchain	offset X'0' in ONCA
BOS Beginning of segment	offset X'8' from TCA
Controlled variables	see "Variables"
DCLCB Declare control Block	Deduced from object program listing
DCB	addressed from offset X'14' (20) in FCB
ENVB Environment Block	offset X'c' (12) in DCLCB
DED Data element descriptor	deduced from object program listing
Diagnostic statement table	addressed from offset X'8' from entry point of main procedure
DFB Diagnostic file block	addressed from offset X'40'(64) in TCA
DSA Dynamic storage area	addressed by register 13 (see P3 and D3)
EOS End of segment	offset X'C' (12) in TCA
Event variable	deduced from object program listing and knowledge of parameter lists of I/O and wait modules
FCB File control block	identified in PL/I dumps. Addressed via PRV and DCLCB
Flow statement table	addressed from offset X'4C' (76) in TCA
Filename	addressed from offset X'10' (16) in FCB
Free-area chain	offset X'8' in implementation-defined appendage, which is addressed from offset X'28' (40) in TCA
Locator/descriptor	deduced from object program listing
LWS Library workspace	addressed from offset X'48' (72) in every DSA
NAB Next available byte	offset X'4C' (76) in DSA
ONCA ON-communications area	the offset of the associated ONCA is held in a halfword at offset X'2' in each section of LWS
ONCB ON-control block start of dynamic ONCB chain	offset X'60' (96) in DSA
- first static ONCB	offset X'5C' (92) in DSA
On-cells	addressed from offset X'70' (112) in DSA
OCB Open control block	deduced from object program listing and parameter list of open module, IBMBOCL
Parameter lists	object program listing and static storage map
Register values	See P3 and O3
RCB Request control block	object program listing and static storage map
SIOCB Stream I/O control block	object program listing
Symbol table	Static listing
Symbol table vector	Static listing
Statement number table	See Diagnostic statement table
Static storage	addressed by register 3 in compiled code. See

	P3 and O3
Segment number	first two bytes of BOS, EOS, or NAB. 'FF'=1,'FE'=2 etc.*
Tasking Appendage	addressed from X'2C' (44) in the TCA.
Task variable	addressed from X'24' (36) in the TCA.
TCA Task communications area	addressed by register 12. See P3 and D3
Variables	
automatic	offset from DSA of block in which they are declared. As shown in variables offset map. See V1.
based	address of the pointer must be deduced from the object program listing. This gives the address of the variable. See V2
controlled	PRV offset referenced in compiled code holds latest allocation of the variable. A chain-back through the previous allocation can be made using the header chain. See V3
static	offset from register 3 is shown in variables offset map. See V4.
area	as for other variables depending on storage class. See V5
Variables in areas	find address of area. Find variable from offset within areas shown in compiled code. See V6

*When the first two bytes of EOS and BOS are greater than two bytes of NAB, it means that an extra segment of storage has been used, but not yet freed. See chapter 6, "Storage Management."

Special Considerations for Multitasking

The major difference between a dump of a multitasking program and the dump of any other PL/I program is that certain relevant items are held within the control task. For this reason, the control task is always dumped as well as the current task.

The contents of the dump of a tasking program depend on the dump options specified. If A (all) is used all the tasks will be dumped. If O (only current task) is specified the control task and the current task will be dumped.

The dump is carried out within the control task and this prevents access to the tasking housekeeping during the execution of the dump. However, this does not prevent access by other tasks to PL/I variables which may be dumped. Subtasks of the current task can access and alter values within the ISA of the current task. Consequently the values of the variables printed cannot be guaranteed to be those that were current at the invocation of the dump.

As explained in chapter 14, the DSA chaining differs slightly when a program is multitasking. The backchain passes through the dummy DSA of the task and ends at the dummy DSA of the major task. The DSA of the block in which the task was attached is not included in the backchain.

Compiled code and the static control sections generated by the compiler are always held in storage associated with the control task.

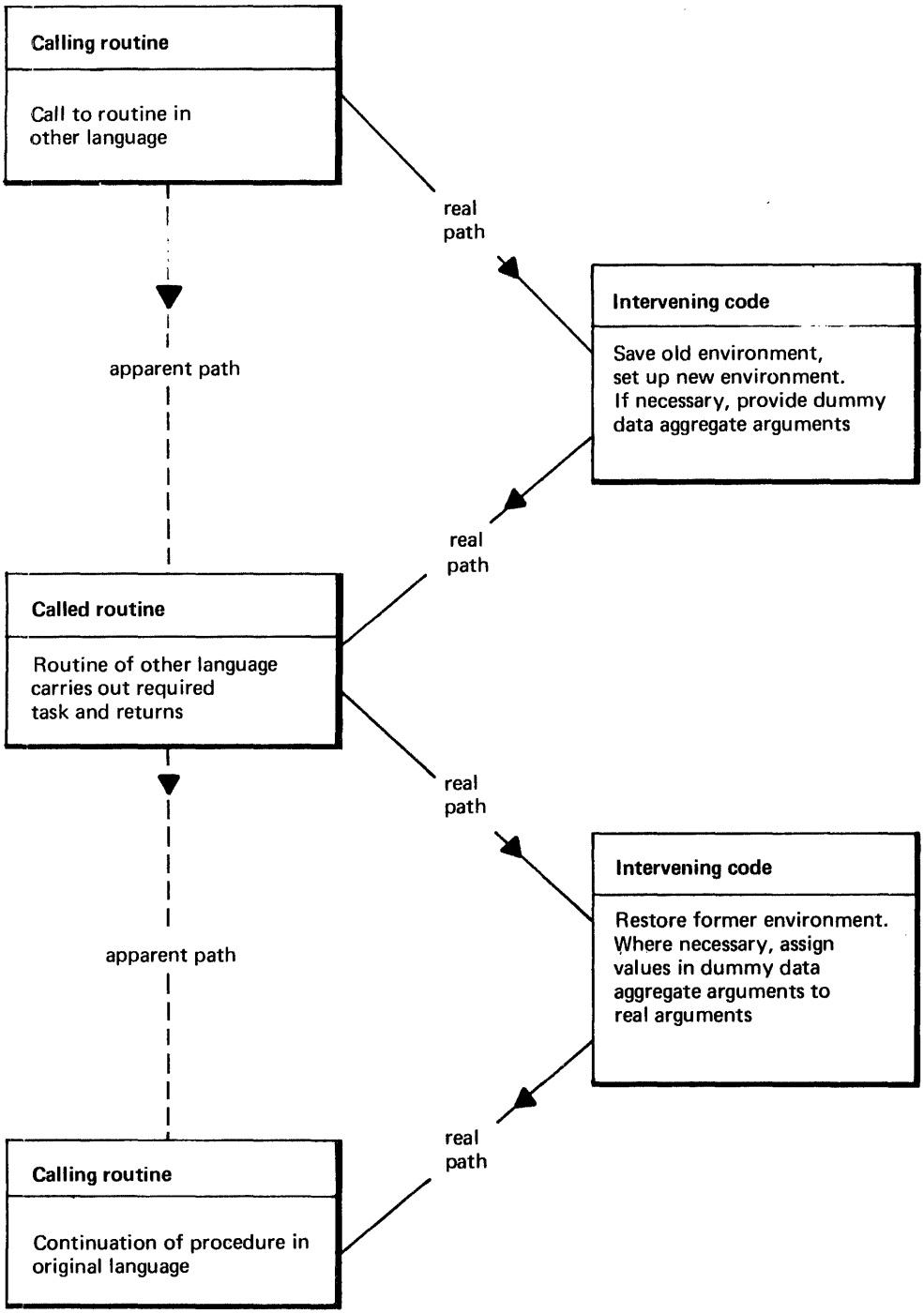


Figure 13.1. The principles of interlanguage communication

Chapter 13: Interlanguage Communication

The OS PL/I Optimizing Compiler allows subroutines compiled on IBM OS COBOL or FORTRAN compilers to be used in PL/I programs compiled on the optimizing compiler. Similarly, it compiles PL/I programs that can be run as subroutines of either COBOL or FORTRAN programs.

Facilities are also provided to overcome the addressing problems when passing arguments to assembler language routines. These are described under the heading "ASSEMBLER Option" later in this chapter.

A full description of how to use the interlanguage communication facilities is given in the language reference manual for this compiler. A detailed description of the PL/I library routines involved is given in the resident library PLM. This chapter explains the basic design principles used and will assist in understanding the situation in main storage during the execution of a program involving interlanguage calls.

The interlanguage facilities are summarized below for background information.

Summary of Interlanguage Facilities

The interlanguage facilities allow any number of calls to be made, and calls to both COBOL and FORTRAN routines can be made in the same program. PL/I can call COBOL that calls PL/I that calls FORTRAN; FORTRAN can call PL/I that calls COBOL, and so on. All calls must, however, be made either to PL/I or from PL/I. Calls cannot be made directly between COBOL and FORTRAN. Options allow the programmer to specify that PL/I interrupt-handling facilities will be available through the COBOL or FORTRAN routines for those program checks that are not handled by COBOL or FORTRAN. Options also allow the programmer to specify whether he wishes data aggregates to be automatically re-formatted when passed as arguments. (The programmer may wish to carry out the re-formatting himself.)

The language involved is fully described in the language reference manual. Briefly, it is as follows. For a PL/I procedure to call a COBOL or FORTRAN routine, the name of the routine must be declared as an external entry point with the option COBOL or FORTRAN in the OPTIONS attribute. If the programmer wishes to take advantage of the PL/I error-handling or interrupt-handling facilities in a COBOL or FORTRAN routine,the INTER option must be included in the declaration. When a PL/I procedure is to be called by COBOL or FORTRAN, the keyword COBOL or FORTRAN should be included in the OPTIONS option of the PROCEDURE or ENTRY statement. To override the creation or remapping of dummy arguments for aggregates the options NOMAP, NOMAPIN, and NOMAPOUT can be used.

The compiler also allows the specification of the COBOL option in the ENVIRONMENT attribute of a PL/I file. This is separate from the interlanguage facilities described above, and is a method of allowing data sets produced by programs of one language to be used by programs of the other language. The use of the COBOL option in the ENVIRONMENT attribute is described in the last section of this chapter.

Background to Interlanguage Communication

The major problems involved in allowing procedures written in PL/I to be used with programs written in CCBOL or FORTRAN are:

1. The existence of different data types in the different languages.
2. The different methods of holding data aggregates in the different languages.
3. PL/I's use of locators when passing areas, arrays, strings, and structures as arguments.
4. The different environment required for each language. This consists of :
   a. Different methods of handling program checks and consequently a requirement for the issuing of new SPIE macro instructions when a new language is entered.
   b. The dependence of PL/I and FORTRAN on initialization and termination routines to set up and discard their environments.

The first of these problems must be solved by the programmer himself, by ensuring that

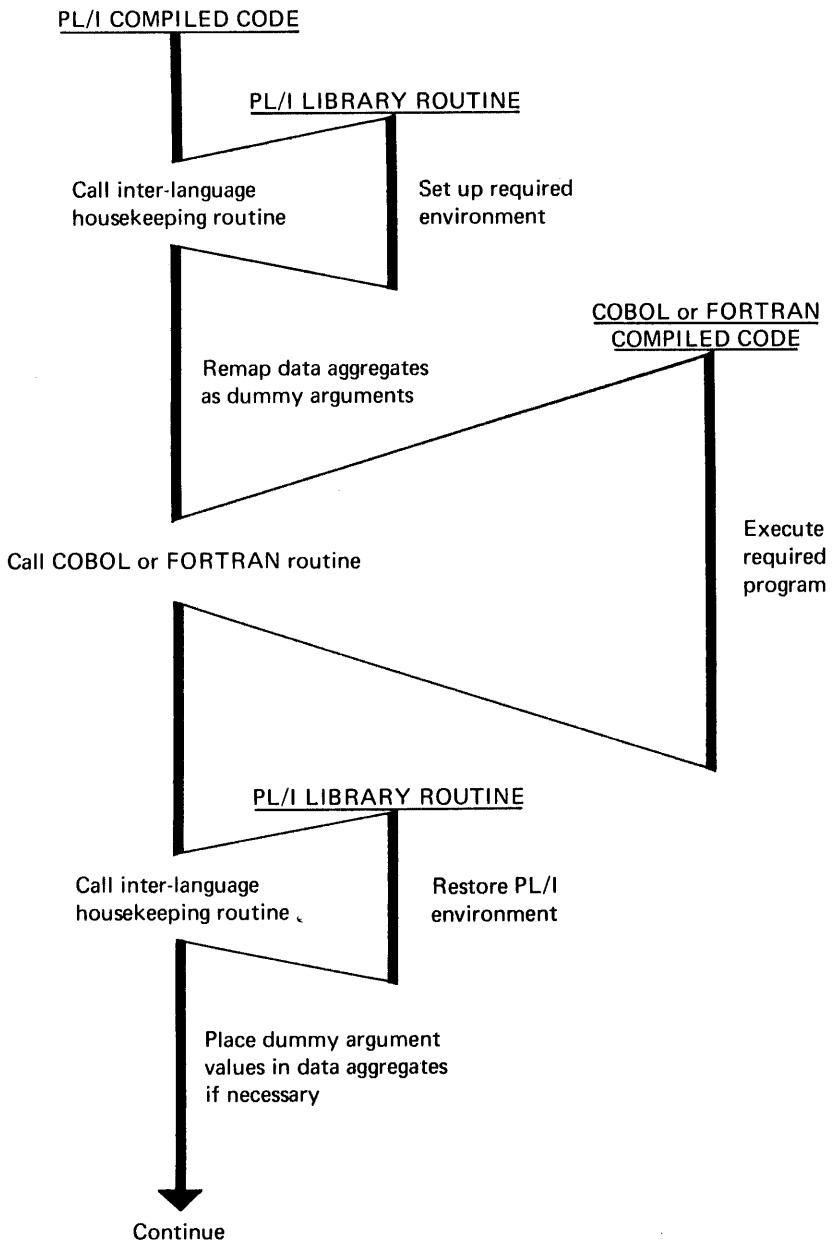


Figure 13.2. Calling sequence when PL/I calls COBOL or FORTRAN

arguments passed between the routines are of suitable data types. (Information in the language reference manual for this compiler enables the programmer to do this.)

The other problems mentioned above are handled automatically by the interlanguage communication facilities of the compiler. They are summarized below.

DIFFERENCES IN DATA AGGREGATES

Structures in PL/I and COBOL, and arrays in PL/I and FORTRAN, are held in different manners.

COBOL structures are mapped as they are declared, with the structure starting on a doubleword boundary and each item separately aligned. PL/I structures are mapped in a manner that minimizes padding.

In FORTRAN, multidimensional arrays are held in column-major order. In PL/I, they are held in row-major order. Thus the second element in a FORTRAN two-dimensional array has the subscript (2,1), whereas the second element in a PL/I two-dimensional array has the subscript (1,2).

Structures are not available in FORTRAN. COBOL data with the OCCURS option, which can be equivalent to PL/I arrays, is held in row major order, as are PL/I arrays.

USE OF LOCATORS

When passing arguments, PL/I passes the address of locators for areas, arrays, strings, and structures rather than the address of the items themselves. This is because the routine that receives the arguments may require information about bounds or sizes of the data passed, and this is accessible through the locator. Other languages, however, expect the address of the data to be passed. The correct type of paramter list must therefore be set up when an interlanguage call is made.

DIFFERENCES OF ENVIRONMENT

PL/I, COBOL and FORTRAN all have different methods of handling program checks. PL/I allows the programmer to handle all program checks. FORTRAN allows the programmer to handle certain program checks. COBOL leaves program checks almost entirely in the hands of the system. Because of the different requirements, a new SPIE macro instruction must be issued whenever control passes between languages. The INTER option demands that program checks are analyzed when they occur and that they are passed to the appropriate language. If they are to be passed to PL/I, the PL/I environment must be restored. For these reasons the INTER option demands that further SPIE macro instructions be issued.

IBM FORTRAN compilers and the PL/I optimizing compiler rely upon initialization routines to set up an environment in which the compiled code routines can operate. In FORTRAN, the main task of the initialization routine is to issue a SPIE macro instruction to initiate the FORTRAN error-handling scheme. In PL/I, the initialization routines prepare for the PL/I error-handling schemes and also prepare the way for dynamic storage allocation. During PL/I initialization routines, register 12 is pointed at the TCA, which is used for addressing a number of housekeeping fields and library routines. Register 13 is pointed at a DSA which contains a standard save area, a NAB pointer pointing to the next available byte of last-in, first-out dynamic storage, various other housekeeping fields, and storage for variables declared automatic. (See chapter 1 and chapter 5 for a discussion of the PL/I environment.)

When PL/I is called from either COBOL or FORTRAN the PL/I environment must be set up before the program can be run. Similarly, when PL/I calls another language, the environment suitable for the program that has been called must be set up, and the PL/I environment saved so that it may be restored on return to PL/I.

THE PRINCIPLES OF INTERLANGUAGE COMMUNICATION

Figure 13.1 shows the method used to handle interlanguage communication problems. Interface code is inserted immediately before and immediately after the execution of a routine in a different language. This code saves the existing environment and sets up the required environment. Where necessary it creates dummy aggregate arguments of the correct format. The interface code is divided between compiled code and library routines. Compiled code handles data aggregate arguments and calls a library routine to handle the problems of environment. Three PL/I resident library routines are used; one for calls to each

language. These routines are known as the interlanguage housekeeping routines.

The interface code is always placed in PL/I, because it is the PL/I compiler that manages the interlanguage facilities. However the position of the code depends on whether PL/I is the called or calling program.

PL/I Calls COBOL or FORTRAN

When the calling program is PL/I the interface code is placed immediately before and immediately after the call to the COBOL or FORTRAN routine. The sequence, is shown in figure 13.2 and is summarized below.

1. Compiled code remaps data aggregate arguments if necessary.
2. Compiled code calls the interlanguage housekeeping routine, which handles environment problems.
3. Compiled code calls the COBOL or FORTRAN routine.
4. On return from the COBOL or FORTRAN routine, compiled code calls the interlanguage housekeeping routine to restore the PL/I environment.
5. Compiled code re-maps dummy data aggregate arguments if any, and continues.

The code generated by the compiler is shown in figure 13.3.

FORTRAN or COBOL Calls PL/I

When the called program is PL/I, the necessary interface code is placed at the start and finish of the PL/I program. The interface code is compiled as an encompassing routine to the required PL/I routine.

The method used, is to compile the PL/I program in the normal way except that it is compiled as internal to an interface procedure that contains the interface code.

This interface, or encompassing procedure is given the external name of the PL/I procedure and is thus called by the other-language routine. The interface procedure, when it has called the interlanguage housekeeping routine and handled the data aggregate arguments, calls the required PL/I routine. Control returns to the original caller by way of the interface routine which again handles the interlanguage problems before returning.

The sequence of events when PL/I is the called program is shown in figure 13.3 and is summarized below.

1. A COBOL or FORTRAN routine calls the PL/I routine.
2. Control passes to the interface routine which has been compiled with the ESD name of the PL/I routine or entry point.
3. The interface routine calls the interlanguage housekeeping routine to handle environment problems.
4. The interface routine handles data aggregate arguments as necessary.
5. The interface routine calls the required routine.
6. Control returns from the required routine to the interface routine. The interface routine handles data aggregate arguments as necessary.
7. The interface routine calls the interlanguage housekeeping routine to handle environment problems.
8. Control returns from the interface routine to the original caller.

Retaining the Environment

The overhead of setting up PL/I and FORTRAN environments every time a routine is called could become considerable if the routine were called a large number of times. To prevent this overhead, the environment is retained until the routine that calls the other language routine is itself terminated. This is done by a rearrangement of the save area chaining, so that the PL/I and FORTRAN termination routines are not entered until the calling program is itself terminated.

The arrangement introduces certain housekeeping problems which are resolved by inserting further save areas into the chain. These save areas have register 14 values that result in control being passed to subroutines of the interlanguage housekeeping routines. These subroutines, known as tail code, handle problems such as preserving values passed from the caller to the caller's caller.

```
                         SOURCE

 1          P13P2:PROC;
 2    1              DCL FRED OPTIONS(COBOL),
                         1 STRUCTURE,
                             2 C CHAR (1),
                             2 D FIXED BINARY (31,0);
 3    1              CALL FRED(STRUCTURE);
 4    1            END;
```

```
* STATEMENT NUMBER  3
000066  41 00 0 008                LA    0,8(0,0)
00006A  58 10 D 04C                L     1,76(0,13)
00006E  1E 01                      ALR   0,1
000070  55 00 C 00C                CL    0,12(0,12)
000074  47 D0 2 018                BNH   CL.4
000078  58 F0 C 048                L     15,72(0,12)
00007C  05 EF                      BALR  14,15
00007E                     CL.4    EQU   *
00007E  50 00 D 04C                ST    0,76(0,13)
000082  41 11 0 000                LA    1,0(1,0)
000086  50 10 D 0A8                ST    1,168(0,13)
00008A  D2 03 D 088 D 0B3          MVC   WKSP.1+16(4),STRUC
                                         TURE.C
000090  58 80 D 088                L     8,WKSP.1+16
000094  D2 03 1 000 D 088          MVC   0(4,1),WKSP.1+16
00009A  58 90 D 0B4                L     9,STRUCTURE.D
00009E  50 90 1 004                ST    9,4(0,1)
0000A2  58 F0 3 00C                L     15,A..IBMBIECA
0000A6  18 71                      LR    7,1
0000A8  05 EF                      BALR  14,15
0000AA  50 70 3 030                ST    7,48(0,3)
0000AE  96 80 3 030                OI    48(3),X'80'
0000B2  1B 55                      SR    5,5
0000B4  41 10 3 030                LA    1,48(0,3)
0000B8  58 F0 3 034                L     15,52(0,3)
0000BC  18 67                      LR    6,7
0000BE  05 EF                      BALR  14,15
0000C0  58 F0 3 010                L     15,A..IBMBIECC
0000C4  05 EF                      BALR  14,15
0000C6  D2 03 D 088 7 000          MVC   WKSP.1+16(4),0(7)
0000CC  58 F0 D 088                L     15,WKSP.1+16
0000D0  D2 03 D 0B3 D 088          MVC   STRUCTURE.C(4),WKS
                                         P.1+16
0000D6  58 60 7 004                L     6,4(0,7)
0000DA  50 60 D 0B4                ST    6,STRUCTURE.D
```

Get VDA for dummy arguments

Place new value in NAB

Move structure into dummy

Branch to interlanguage housekeeping routine

Set up argument list

Branch to COBOL routine

Branch to interlanguage housekeeping routine

Move values from dummy to real arguments

Figure 13.3. Code generated when PL/I calls a COBOL routine

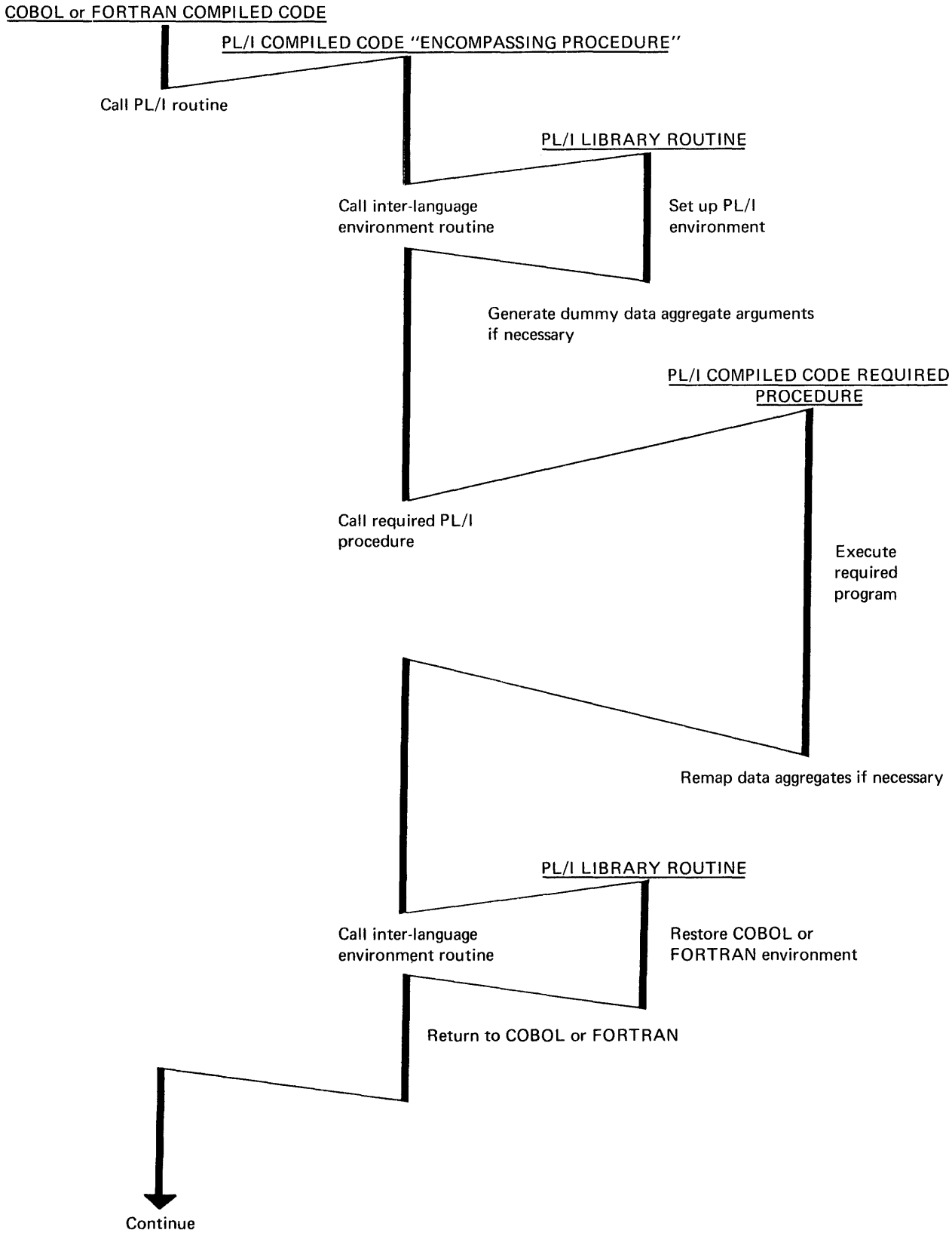
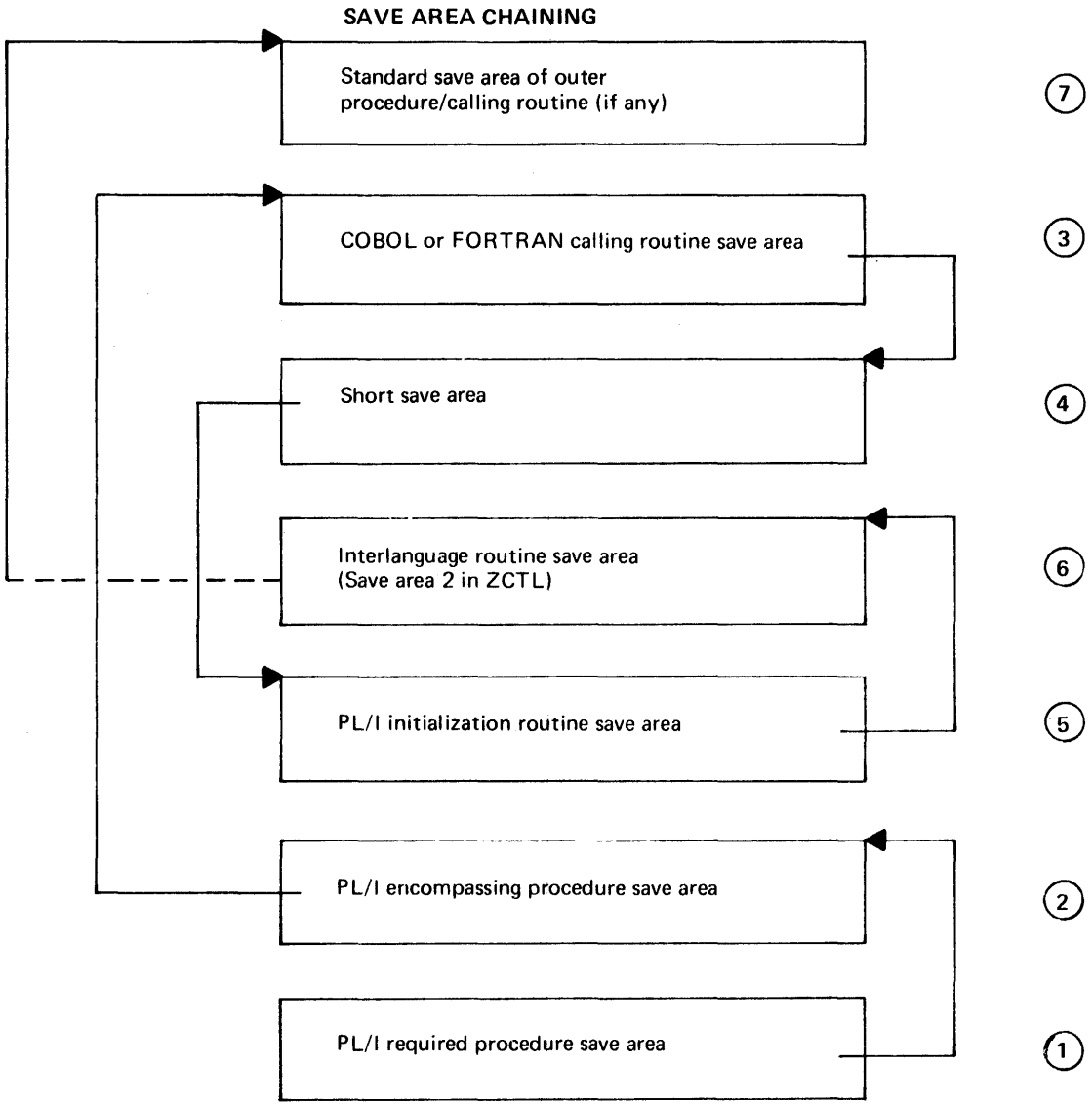


Figure 13.4. The sequence of events when FORTRAN or COBOL calls PL/I



Rearrangement of save area chaining takes place after the first call to PL/I, so that the PL/I environment is not discarded until the calling routine itself is finished.

Save areas that return control to the PL/I initialization routine and interlanguage housekeeping routine are placed before the calling routine. (The numbers 1-7 in the diagram show the order of backchaining).

Figure 13.5. Chaining of save areas when PL/I is called from a COBOL or FORTRAN principal procedure

Handling Changes of Environment

Interlanguage Housekeeping Routines and their Control Blocks

Changes of environment are handled by three resident library interlanguage housekeeping modules, one for calls to each language. Common features are described below. A more detailed description follows for each routine. The routines are:

IBMBIEF for calls to FORTRAN
IBMBIEC for calls to COBOL
IBMBIEP for calls to PL/I

The job of these routines is the saving and restoring of environments. This involves issuing SPIE macro instructions suitable for the called routine and saving the PICA of the calling routine so that a suitable SPIE macro instruction can be issued before return. For PL/I it also involves storing information about dynamic storage allocation, and the TCA address.

The information required when setting up and restoring environments is held in three chained control blocks:

1. IBMBILC1 This is a control section included in every interlanguage housekeeping routine. It contains flags to indicate whether the PL/I, FORTRAN or COBOL environments already exist and, if any do exist, contains a pointer to ZCTL.

2. ZCTL This holds PICA addresses and the TCA address. It is chained to a series of interlanguage VDAs. It also holds flags indicating which languages are currently active.

   ZCTL is generated on the first of a series of interlanguage calls and is retained until that series of calls is completed. For calls to FORTRAN and PL/I it is retained until the routine that made the first interlanguage call is itself terminated.

   Also held in ZCTL are the additional save areas used when the chaining is altered. These are known as save area 1, save area 2, and the ghost save area. The uses of these save areas are given in the individual module descriptions.

3. Interlanguage VDAs These hold flags indicating which languages were active before the latest call was made, the address of the callers PICA, the address of the most recent PL/I DSA.

An interlanguage VDA is acquired for every interlanguage call and discarded when the called routine is terminated. Interlanguage VDAs are held in the PL/I LIFO storage stack.

The methods of chaining used for these control blocks when PL/I is the called and the calling language is shown in figures 13.6 and 13.7. IBMBILC1 contains a pointer to ZCTL and ZCTL contains a pointer to the most recent interlanguage VDA. Interlanguage VDAs hold pointers to previous interlanguage VDAs, if any. If there are none, the pointer field is set to zero.

There is one interlanguage VDA for each interlanguage call. A VDA is set up when the call is made and discarded when the associated routine is terminated. The VDAs hold a record of the ZCTL flags that existed before they were called. These flags are placed in the VDA before the flags are altered and restored in ZCTL when the VDA is discarded. Thus ZCTL always contains a record of the active languages. This information is necessary when handling STOP statements.

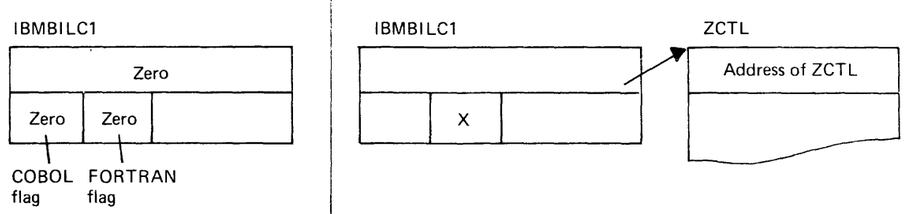
The flags in IBMBILC1 contain a record of the environments that are active. These flags are used to test whether it is necessary to call the FORTRAN or PL/I initialization routines, or whether the environment can be restored from the information saved in ZCTL and the interlanguage VDAs.

Handling FORTRAN and PL/I Initialization/Termination Routines

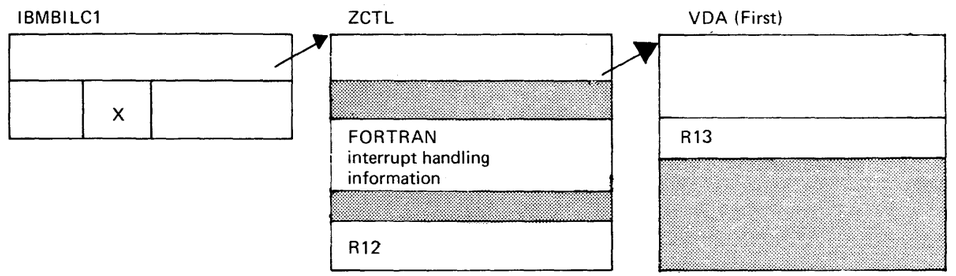
FORTRAN and PL/I environments are set up by initialization routines and discarded by termination routines. To save the overheads of executing these routines on each call to the language the save area for the termination routine is placed above that of the calling program. On the first call the PICA address and, for PL/I only, the current DSA and TCA address are saved. For subsequent calls this information is restored by the interlanguage routines and no call made to the initialization routine. Figure 13.8 shows the principles involved.

The rearrangement of the save area chain results in certain problems, for example returning parameters from the caller to the caller's caller. To overcome these problems additional save areas are inserted in the chain. These save areas result in control passing to subroutines in the interlanguage housekeeping routines known as tail code. Details are given in figures 13.9 and 13.10 and in the individual module descriptions below.

1 **Initial situation**
IBMBILC1 is set up as a control section by the PL/I interlanguage routines. Its first word and flags are initially zero.

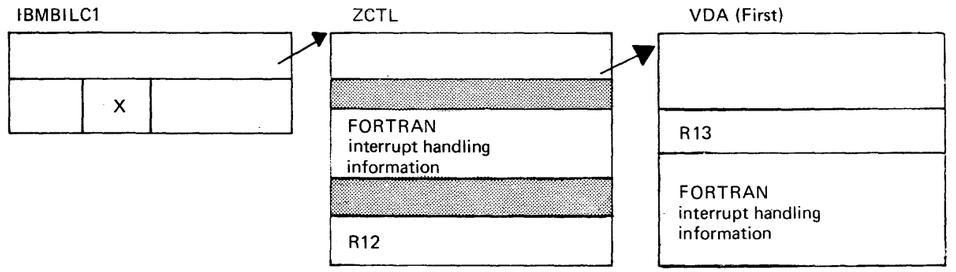


2 **Call FORTRAN from PL/I (IBMBIEF)**
The compiler generates a call to the interlanguage communications routine. This routine:
1. Sets up ZCTL after testing for zero pointer in IBMBILC1. Acquires an interlanguage VDA.
2. Sets ZCTL pointer to interlanguage VDA, and IBMBILC1 pointer to ZCTL.
3. Sets FORTRAN flag in IBMBILC1. Saves R12 in ZCTL, R13 in interlanguage VDA.
4. Calls FORTRAN library to initialize FORTRAN SPIE
5. Resets program check exit as required.
6. Returns to compiled code, which calls FORTRAN procedure.



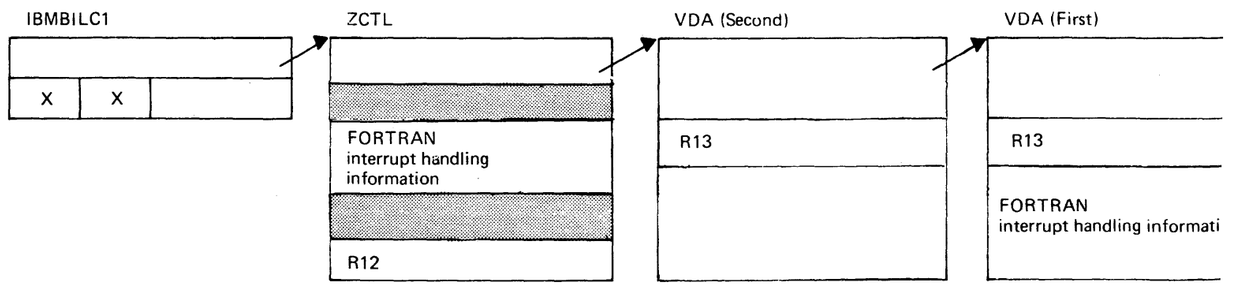
3 **Call PL/I from FORTRAN (IBMBIEP)**
The PL/I program, because it is declared with the option FORTRAN, will have been compiled inside an encompassing procedure. The encompassing procedure is the one called by FORTRAN. The encompassing procedure calls the interlanguage communications routine IBMBIEP, which:
1. Checks IBMBILC1 to see if either FORTRAN or COBOL flag is set. As one flag is set, restores registers.
2. Issues PL/I SPIE and STAE and stores interrupt handling information of calling program in interlanguage VDA.

Control then returns to the encompassing program, which calls the required PL/I program.

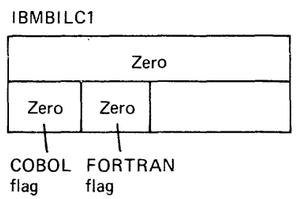


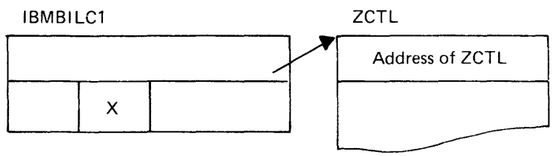
4 **Call COBOL from PL/I (IBMBIEC)**
The PL/I program will contain a call to the interlanguage routine IBMBIEC, which:
1. Sets up another interlanguage VDA, points ZCTL to this VDA, and places the old value of ZCTL's pointer in the VDA.
2. Stores R13 in the new VDA.
3. Issues a SPIE so that error handling will be as requested by PL/I program.

Control is then returned to compiled code, which then calls the COBOL routine.



l section by the PL/I interlanguage
gs are initially zero.





Final situation **8**

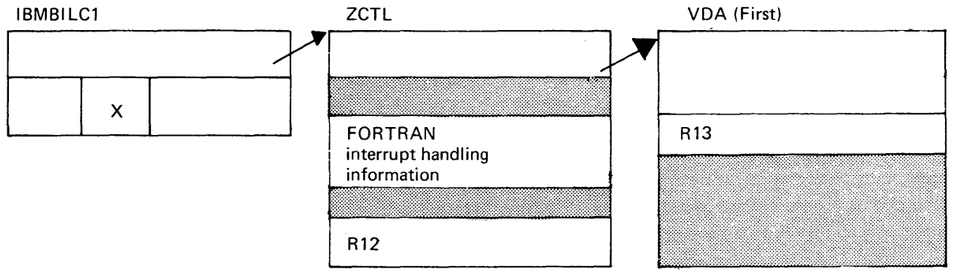
ZCTL is retained until program is completed.

MBIEF)
the interlanguage communications

for zero pointer in IBMBILC1.
VDA.
language VDA, and IBMBILC1

MBILC1. Saves R12 in ZCTL,

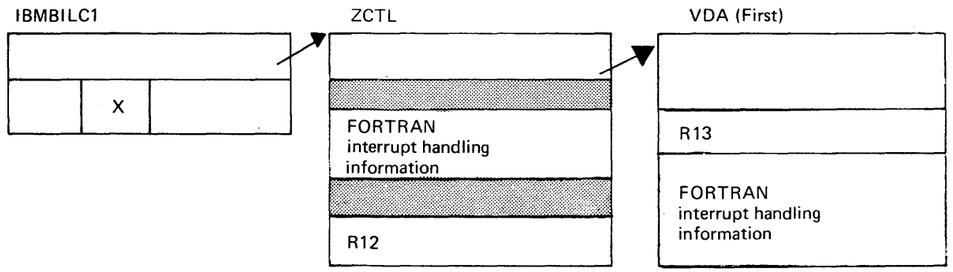
initialize FORTRAN SPIE
as required.
which calls FORTRAN



FORTRAN returns to PL/I (IBMBIEF) **7**

When control returns from a FORTRAN procedure, a call is made to the interlanguage communication routine IBMBIEF, which:
1. Moves the pointer in the VDA to the first word of ZCTL.
2. Issues a PL/I SPIE macro.
3. Issues a PL/I STAE macro leaving the previously-stacked FORTRAN STAE for possible future use.
4. Returns control to compiled code.

MBIEP)
declared with the option
piled inside an encompassing
procedure is the one called by
procedure calls the inter-
ne IBMBIEP, which:
f either FORTRAN or COBOL
t, restores registers.
and stores interrupt handling
gram in interlanguage VDA.
ompassing program, which calls



PL/I returns to FORTRAN (IBMBIEP) **6**

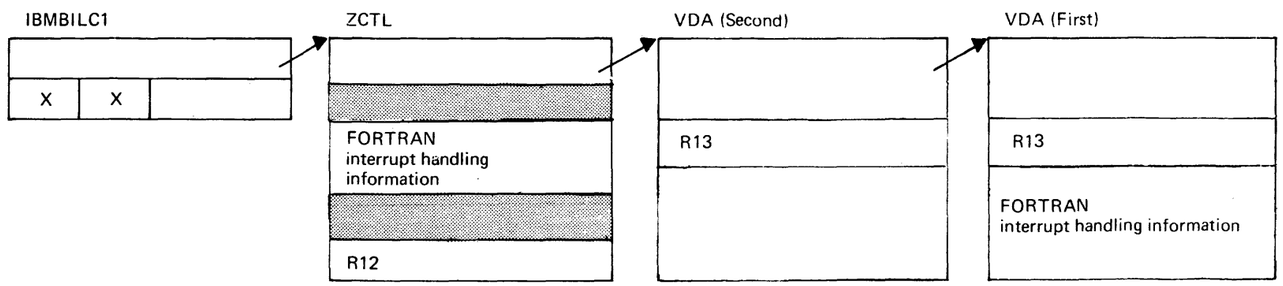
When the required PL/I procedure is finished, it returns control to the encompassing procedure. The encompassing procedure calls the interlanguage routine IBMBIEP, which issues a SPIE macro instruction to restore the error-handling situation to that of the calling routine. The information for the SPIE macro instruction is retrieved from the interlanguage VDA. The current PL/I STAE is canceled, leaving the previously stacked FORTRAN STAE in control.
The interlanguage routine returns control to the PL/I encompas sing procedure, which then returns control to the FORTRAN program.

EC)
call to the interlanguage routine

age VDA, points ZCTL to this
lue of ZCTL's pointer in the

A.
handling will be as requested

piled code, which then calls



COBOL returns to PL/I **5**

The COBOL program returns to the PL/I program, which immediately calls the interlanguage routine IBMBIEC. This routine rearranges the chain by placing the word in the most recent VDA in the first word of ZCTL. It then issues a SPIE macro instruction to restore the PL/I error-handling situation.

PL/I compiled code then continues.

Figure 13.6. Example of chaining sequences (PL/I principal procedure)

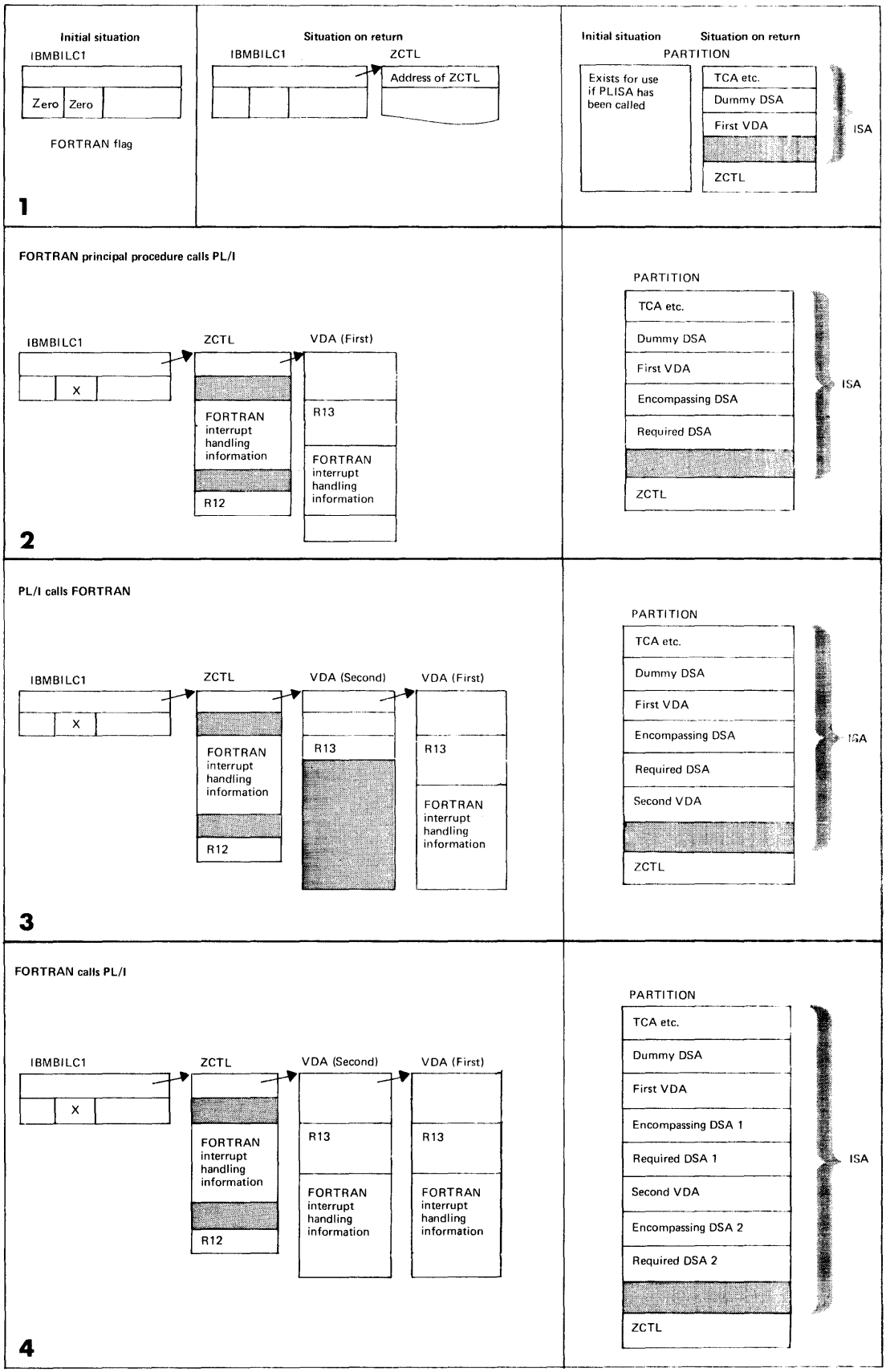


Figure 13.7. Examples of chaining sequences (FORTRAN principal procedure)

Handling the INTER Option

When the INTER option is specified, the programmer gets neither normal PL/I interrupt handling nor the normal interrupt handling for the other language. Instead, he gets PL/I error handling of those interrupts which are left to the system by the non-PL/I languages. To allow for this, the type of interrupt has to be analyzed after it has occurred and passed to the correct error handling routines.

Interrupts are analyzed by subroutines of the interlanguage housekeeping routines known as traps. The interlanguage housekeeping routines save the PICA addresses for the calling language and the called language and issue a SPIE that will pass control to the trap code. When an interrupt occurs control is passed to the trap, which analyzes the interrupt. The trap code issues the appropriate SPIE to restore the required error handling situation and then alters addresses so that normal return from the interrupt will result in control being passed once more to the interlanguage housekeeping trap routine. It then forces an interrupt and the interrupt is handled by the appropriate language. When control returns to the trap a further SPIE is issued so that control returns to the trap should further interrupts occur. The method used for each module is described below.

STOP and STOP RUN Statements

PL/I and FORTRAN STOP statements and COBOL STOP RUN statement cause certain problems because various save areas may be bypassed. The methods adopted to solve these problems are discussed in the individual description of the modules.

Housekeeping Module Descriptions

As the differences between individual interlanguage housekeeping modules are considerable, a detailed description of each module follows. The description covers the following situations:

1. When the associated language routine is called
2. When the associated language routine returns control
3. When an interrupt occurs with the INTER option
4. When a STOP or STOP RUN statement is executed
5. For PL/I and FORTRAN only, when the environment is discarded and the termination routine entered

COBOL WHEN CALLED FROM PL/I (IBMBIEC)

Before Entry to COBOL Program

IBMBIECA - Entry point for COBOL error handling

IBMBIECB - Entry point for INTER error handling

When IBMBIEC is called before the COBOL program, the following must be done:

1. Test to see if this is the first interlanguage call; if so, set COBOL flag in IBMBILC1 and set up ZCTL.
2. Acquire interlanguage VDA and store register 12 and register 13 in the VDA. Write null PICA information in ZCTL.
3. If INTER option not specified (i.e., entry point IBMBIECA), issue SPIE macro instruction so that errors will be handled by the supervisor. Return to compiled code.
4. If INTER option is specified (entry point IBMBIECB), issue new SPIE macro instruction and return so that interrupts will be passed to the trap code.

On Return from COBOL Program (IBMBIECC)

The following actions take place on return:

1. A SPIE macro instruction is executed, which results in the the PL/I error handling scheme being restored.
2. The first word of the interlanguage VDA and the VDA flags are moved into the first word ZCTL, and the VDA is freed.

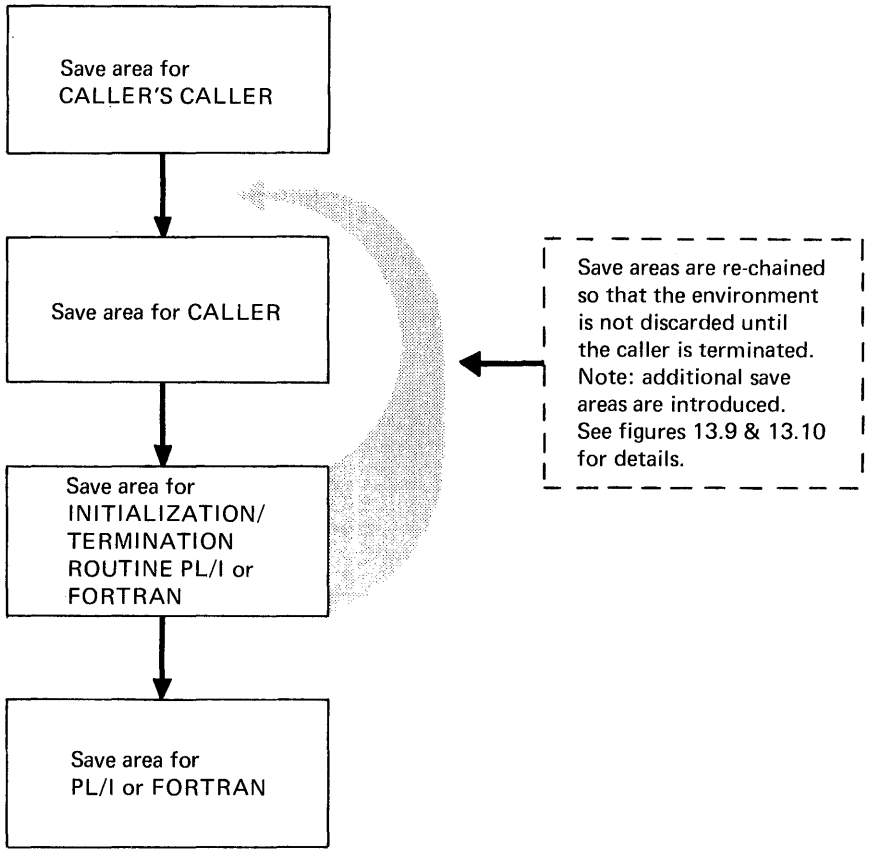


Figure 13.8. The concept of save area rechaining (see figures 13.9 and 13.10 for details)

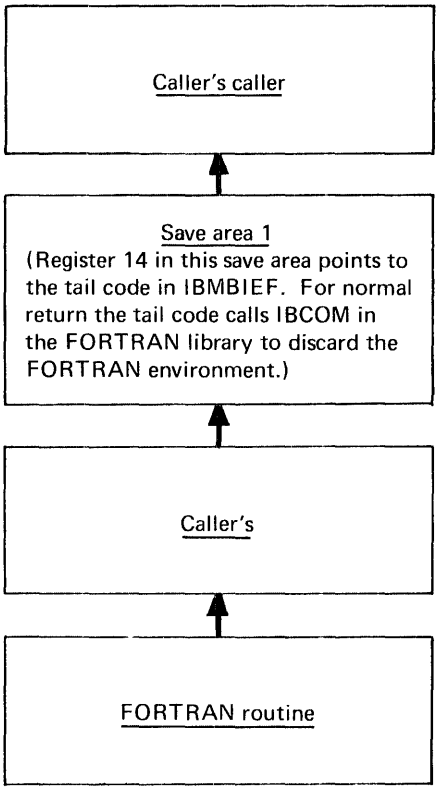


Figure 13.9. Rechaining of save areas when FORTRAN is called from PL/I and the FORTRAN environment needs initializing

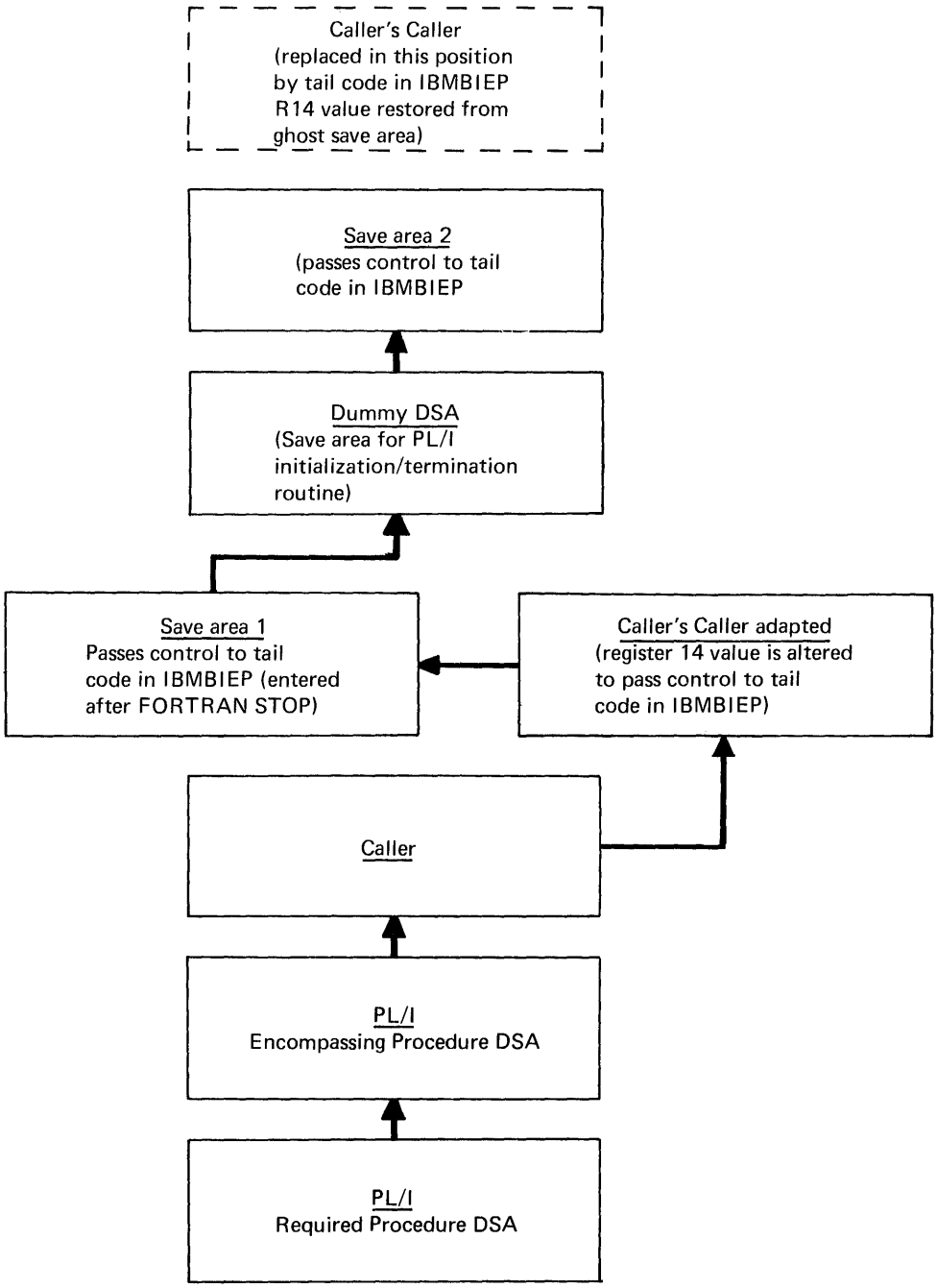


Figure 13.10. Rechaining of save areas when PL/I is called from FORTRAN or COBOL and the environment requires initialization

Action on Interrupt in COBOL with INTER

If the INTER option is not specified, all program checks will be handled by the supervisor in the usual manner. If the INTER option is specified and the program has been compiled with a request for the COBOL interrupt handler not to be called, the following takes place.

1. During the first invocation of IBMBIEC, a SPIE macro instruction is issued, which results in interrupts being passed to entry address in the trap.

2. When an interrupt occurs, register 12 and register 13 are restored thus restoring the PL/I environment.

3. A DSA is acquired for IBMBIEC in LWS. The address of the interrupt, in the second word of the PSW, is saved in this DSA and replaced by the address of another entry address in the trap. For underflow interrupts, the four bytes preceding the point of interrupt are also copied and placed before the trap in case the error handler needs to examine them. The trap acts as the return address for the PL/I error handler.

4. Flags are set in the TCA and DSA to indicate that it is possible for an abnormal GOTO to occur in a PL/I on-unit.

5. A SPIE macro instruction is issued to transfer the program check exit to the PL/I error-handling routines whose address is held in the TCA appendage.

Return from interrupt: If there is a GOTO out of a PL/I on-unit, control passes to the abnormal GOTO subroutine, this is because flags indicating an abnormal GOTO situation are set up by the trap code. The abnormal GOTO subroutine analyzes these flags and passes control to IBMBIEC which handles any necessary housekeeping problems.

If the return is normal, the PL/I error handling routines return control to the address in the second word of the PSW. This word has been altered by code in the trap, and further trap code in IBMBIEC is entered.

It is necessary to return to the point of interrupt in the COBOL program without changing any of the register values and this can only be done via the supervisor. A new SPIE is set to point to further trap code and an interrupt forced. The program is now in an interrupted state, the original INTER SPIE is reissued, and the registers and PIE are restored. The original interrupt address is set in the PSW. Control is returned to the supervisor which passes control to the address in the PSW with the correct register values restored.

Zerodivide On-Units

When used with certain COBOL compilers, normal return from a zerodivide on-unit will result in a data exception. This is because a ZAP instruction is executed after the divide on computational-3 data. The ZAP instruction picks up an invalid field.

Handling STOP RUN statements

ANS COBOL STOP RUN statements are handled by a COBOL routine which passes control to a specified address. When IBMBIEC is called before entry to a COBOL program this address is set to the tail code in IBMBIEC. This tail code dechains all save areas or routines that were entered after the PL/I caller and then executes a PL/I STOP statement.

FORTRAN WHEN CALLED FROM PL/I (IBMBIEF)

When FORTRAN is called by PL/I, IBMBIEF is entered immediately before and immediately after the execution of the FORTRAN program. The processing done before entry to the FORTRAN program depends on whether the INTER option is specified. Entry point IBMBIEFA handles calls without the INTER option. Entry point IBMBIEFB handles calls with the INTER option.

Before Entry to the FORTRAN Program

Entry Point IBMBIEFA FORTRAN error handling

Entry Point IBMBIEFB INTER error handling

Before the call to FORTRAN, IBMBIEF does the following:

1. Tests the flags in IBMBILC1 to discover if this is the first interlanguage call. If it is the first call, it sets up ZCTL and sets the FORTRAN flag in IBMBILC1. If it is not the first call, it tests to see

whether the FORTRAN flag is set in IBMBILC1 and sets the FORTRAN flag if it is not already set.

2. IBMBIEF stores register 13 in the interlanguage VDA, thus saving the PL/I environment.

3. If the FORTRAN environment has not previously been set up, calls the FORTRAN initialization routine. This routine sets up the program check exit so that program interrupts will be handled by the FORTRAN error handling method. The FORTRAN error data is stored in ZCTL. Save area one (SA1) is then inserted in the save area chain. The resulting save area chaining is shown in figure 13.9.

4. IBMBIEF acquires an interlanguage VDA. Points the first word of ZCTL to this VDA, taking the value previously in the first word of ZCTL and placing it in the first word of the VDA. (This places the new VDA at the head of a chain starting from ZCTL.)

5. If INTER option is not specified, issues a FORTRAN SPIE macro instruction from ZCTL, sets program mask to '2', and returns to compiled code.

6. If INTER option is specified, a SPIE macro instruction is issued that will result in control being passed to the trap should an interrupt occur. The program mask is reset to 'E' in case it was changed by the FORTRAN initialization routine.

Action on Return from FORTRAN Program (IBMBIEFC and IBMBIEFD)

When return is made from the FORTRAN subroutine, PL/I compiled code immediately makes a call to the FORTRAN interlanguage routine. If the FORTRAN routine may have been used as a function, entry point IBMBIEFD is used. Otherwise, entry point IBMBIEFC is used. The module IBMBIEF does the following:

1. A SPIE macro instruction is issued that resets the program check exit to the PL/I error-handling modules, and the program mask is set to 'E'.

2. The first word of the interlanguage VDA is placed in the first word of ZCTL. The VDA flags inserted in ZCTL and the VDA freed.

3. For entry point IBMBIEFD (the FORTRAN function entry point) the parameter list passed by PL/I is examined, and the values are moved from registers in which they were placed by the FORTRAN routine, to the location expected by PL/I.

Action on Interrupt in FORTRAN

If the INTER option is not specified, the action on any interrupt that occurs in the FORTRAN program will be that specified in the FORTRAN error-handling scheme. However, if the INTER option is specified, all program checks that are not handled by FORTRAN error-handling are passed to the PL/I error-handling modules.

The FORTRAN error-handling scheme is used after the following interrupts have occurred:

1. Specification (other than for invalid instruction address)

2. Fixed-point divide

3. Decimal divide

4. Exponent overflow

5. Exponent underflow

6. Floating-point divide

All other program checks are handled by the PL/I error handler.

If the INTER option is specified, when an interrupt occurs the following takes place:

1. When control is passed by the supervisor to the trap, the type of interrupt is discovered by examining the PSW. If the interrupt is one of the types that can be handled by FORTRAN, the normal FORTRAN environment is established and the FORTRAN error handling module invoked.

2. If it is not the type of interrupt that can be handled by FORTRAN, register 12 is restored from ZCTL and 13 from the latest interlanguage VDA, thus restoring the PL/I environment.

3. The address of the interrupt is taken from the second word of the PSW and stored in the DSA. The second word of the PSW is then replaced by an entry address in the trap in IBMBIEF.

4. Flags are set in the TCA and DSA to indicate that it is possible for an

abnormal GOTO to occur in a PL/I on-unit.

5. A SPIE macro instruction is then issued to restore the PL/I error-handling situation. A branch is then made to the PL/I error handler.

Return from interrupt: If there is a GOTO out of a PL/I on-unit, control passes to the abnormal GOTO subroutine, this is because flags indicating an abnormal GOTO situation are set up by the trap code. The abnormal GOTO subroutine analyzes these flags and passes control to IBMBIEF which handles any necessary housekeeping problems.

If the return is normal, the PL/I error handling routines return control to the address in the second word of the PSW. This word has been altered by code in the trap, and further trap code in IBMBIEF is entered.

It is necessary to return to the point of interrupt in the FORTRAN program without changing any of the register values and this can only be done via the supervisor. A new SPIE is set to point to further trap code and an interrupt forced. The program is now in an interrupted state, the original INTER SPIE is reissued, and the registers and PIE are restored. The original interrupt address is set in the PSW. Control is returned to the supervisor which passes control to the address in the PSW with the correct register values restored.

Termination of Caller

When the routine that called FORTRAN is terminated, control is passed to the address held in the register 14 save area in save area one. This address is the address of the tail code in IBMBIEF. If the return is normal, the tail code calls IBCOM in the FORTRAN library to discard the FORTRAN environment. It then frees ZCTL and returns control to the caller's caller.

STOP Statements

If control returns to the tail code because of a FORTRAN STOP statement the tail code discards any save areas that may have been bypassed by the FORTRAN STOP statement and finally executes a PL/I STOP statement which terminates the program.

PL/I CALLED FROM COBOL OR FORTRAN (IBMBIEP)

As with the other interlanguage communication routines, IBMBIEP is called immediately before and immediately after the program that is to be executed. However, the interlanguage housekeeping routine cannot be called direct from the COBOL or FORTRAN routine, because the existence of such a routine is unknown to COBOL or FORTRAN. To overcome this problem, an encompassing routine is generated with the same entry name as the PL/I routine. This encompassing routine is called by COBOL or FORTRAN and in turn calls the interlanguage housekeeping routine and the required PL/I routine.

Although the names of both PL/I procedures are the same, the encompassing routine gets control when called from COBOL or FORTRAN. This happens because no ESD records are generated for the interlanguage entry points of the required PL/I program. Code for a PL/I encompassing routine is shown in figure 13.5. Figure 13.4 shows the calling sequence.

Before Entry to PL/I Program (IBMBIEPA)

Before a call is made to the required PL/I program, IBMBIEP does the following:

1. Tests to see if the PL/I environment has already been initialized, by examining whether the COBOL or FORTRAN flag in IBMBILC1 is set.

2. If the COBOL of FORTRAN flag is set, this means that a previous interlanguage call has been made, and as the call must have been made either to or from PL/I, the PL/I environment must have been set up. If it is established that the PL/I environment exists, register 12 is restored from ZCTL. A SPIE macro instruction is issued so that program checks are handled by the PL/I error handler. The address of the old PICA is stored in the interlanguage VDA. Control returns to the encompassing routine.

3. If neither the COBOL nor the FORTRAN flag is on, PL/I is being called for the first time by a procedure in a program whose principal procedure is COBOL or FORTRAN. The following action is taken:

   a. IBMBIEP issues a GETMAIN macro instruction and sets up ZCTL in the storage acquired.

b. The PL/I initialization routine, IBMBPIR is called. It sets up the PL/I environment and returns control to an address in IBMBIEP that it was passed by IBMBIEP. IBMBIEP then stores the registers of IBMBPIR in the dummy DSA.

c. The chaining of save areas is then altered, so that the dummy DSA (the save area used by IBMBPIR) is above the calling program's standard save area. The result of this is that, when the encompassing routine is complete, return is made to the COBOL or FORTRAN calling routine rather than to IBMBPIR. Thus the PL/I termination routine is not entered and the PL/I environment is retained until the COBOL or FORTRAN calling program is completed. Two further save areas are also inserted into the chain. These result in control being passed to tail code in IBMBIEP, which handles housekeeping problems. The save area of the caller's caller is also altered so that the register 14 value also points at tail code in IBMBIEP. The true register 14 value is saved in ZCTL in storage known as the ghost save area. The resulting save area chain is shown in figure 13.10. Action taken when the calling routine is terminated is described below, under the heading "Termination of PL/I Environment".

4. A DSA for the encompassing routine is acquired.

5. The address of the new DSA is placed in the register 0 slot of the dummy DSA.

6. Control is then returned to compiled code in the encompassing routine.

Action after the PL/I Program is Completed

Entry point IBMBIEPC - normal

Entry point IBMBIEPD - return value expected

IBMBIEP is called at the end of the PL/I routine by the encompassing routine generated by the compiler. If the calling program is FORTRAN, a returned value may be expected in register 0 or one or more of the floating-point registers. When a returned value may be required, the entry point IBMBIEPD is used and the returned value is loaded into the required position. In other situations, the entry point IBMBIEPC is used. The module resets the program mask by issuing SPIE macro instruction to restore the calling routine's program check exit, the address of which has been stored in the interlanguage VDA.

Interrupt Handling

When PL/I is called by COBOL or FORTRAN, error handling is carried out in the normal PL/I manner. The SPIE macro instruction is issued by IBMBPII when the PL/I environment is first set up. For calls after the first, the SPIE macro instruction is issued by IBMBIEP.

Termination of PL/I Environment

The PL/I environment is discarded when the caller's caller is terminated. In a normal situation control is returned by the caller to the address held in the register 14 save area of the caller's caller. This address was altered during the initialization of the PL/I environment to point to tail code in IBMBIEP. This code receives control and rearranges the save area chaining. It then returns to IBMBPIR whose registers are in the dummy DSA. The PL/I program is then terminated and control returns to save area 2. This again points to tail code in IBMBIEP. This tail code restores the correct register 14 value of the caller's caller from the ghost save area and returns to the caller's caller.

STOP and STOP RUN Statements

For a PL/I STOP statement the action is carried out in a normal manner and flags in save area one indicate that an abnormal GOTO situation exists. The situation is analyzed by the abnormal GOTO subroutine and control is passed to tail code whose address is held in save area one.

For a FORTRAN STOP statement when the calling program is FORTRAN the situation depends on how many levels of FORTRAN precede PL/I. If the caller is the highest level of FORTRAN prior to PL/I, control will be passed to save area one and tail code entered to carry out the necessary housekeeping. If there is more than one

level of FORTRAN, control will pass to the highest active level of FORTRAN and the job will be terminated without carrying out PL/I program termination.

A COBOL STOP RUN statement will be analyzed by IBMBIEC which will execute a PL/I STOP statement.

Handling Data Aggregate Arguments

In order to communicate effectively between COBOL and PL/I, and FORTRAN and PL/I, a method of handling data aggregate arguments is necessary, because the three languages hold data aggregates in different ways.

ARRAYS

Arrays as such are not used in COBOL. The use of OCCURS in structures does, however, have a similar effect. However, PL/I structures of arrays and COBOL structures using OCCURS are both held in row-major order. In FORTRAN, arrays are held in column-major order. Thus, in a two-dimensional array, the element known in the FORTRAN array as (2,1) will become (1,2) in the PL/I array.

STRUCTURES

Structures are not used in FORTRAN. In COBOL the alignment requirements are met differently from PL/I. Full details of the differences in mapping are given in the language reference manual for this compiler.

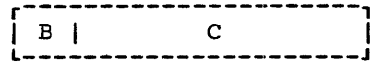
COBOL structures are mapped as follows. Working from the start, each item is aligned to its required boundary in the order in which it is declared, the structure starting on a doubleword boundary.

PL/I structures are mapped by a method that minimizes the unused bytes in the structure. Basically, the method used is first to align items in pairs, moving the item with the lesser alignment requirement as close as possible to the item with the greater alignment requirement. The method is described in full in the language reference manual.

Take, for example, a structure consisting of a single character and a fullword fixed binary item. The fullword binary item has a fullword alignment requirement; the character has a byte alignment requirement. In PL/I, the structure would be declared:

```
DCL 1  A,
    2  B CHAR (1),
    2  C FIXED BINARY (31,0);
```

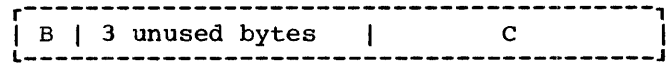
and would be held thus:



In COBOL, the structure would be declared:

```
01 A.
   02  B,  PICTURE X, DISPLAY.
   02  C,  PICTURE S9(9), COMPUTATIONAL.
```

and would be held thus:



METHODS USED TO HANDLE DATA AGGREGATE ARGUMENTS

The method used in handling data aggregates is to create dummy arguments of the correct format and let the called routine use the dummy. The values in the dummy are then assigned to the original argument when the execution of the called program is completed.

If the data aggregates are not adjustable, the mapping will be done during compilation and both the PL/I and the COBOL or FORTRAN mapping are produced. If the data aggregates are adjustable, the mapping is done during execution. Before the execution of the call to a program in another language, the data is transferred into the correctly mapped aggregate, which will be held in PL/I temporary storage. The values are reassigned to the original data aggregate after execution of the interlanguage program.

The assignment of data between the dummy and the argument is done by compiled code.

NOMAP, NOMAPIN, AND NOMAPOUT OPTIONS

The NOMAP, NOMAPIN, and NOMAPOUT options can be used by the programmer to specify that data aggregates will not be remapped and placed in dummy arguments.

When NOMAP is specified, or when both NOMAPIN and NOMAPOUT are specified, the

dummy is not generated at all, and the structure or array is passed as it stands.

When only NOMAPIN is specified, a dummy is created, but it is not initialized with the values of the aggregate being passed. However, on return from the COBOL or FORTRAN routine, the data in the dummy is placed in the data aggregate that is being passed.

When only NOMAPOUT is specified, a dummy is created, and the data from the data aggregate is moved into the dummy. When control is returned to the calling program, however, the data from the dummy is not moved into the data aggregate that was passed.

CALLING SEQUENCE

When PL/I calls COBOL or FORTRAN passing data aggregates as arguments, the sequence of events is:

1. Handle data reassignment to dummy by compiled code.
2. Call interlanguage housekeeping routine.
3. Call COBOL or FORTRAN routine.
4. Call interlanguage housekeeping routine.
5. Assign data in dummy to real argument, by means of compiled code.

When COBOL or FORTRAN calls PL/I, the sequence of events is:

1. The COBOL or FORTRAN routine calls the encompassing PL/I routine.
2. The encompassing PL/I routine:
   a. Calls the interlanguage housekeeping routine.
   b. Sets up the necessary dummy data aggregate argument by compiled code.
   c. Calls the required PL/I routine.
   d. Reassigns the data from the dummy by compiled code.
   e. Calls the interlanguage housekeeping routine.
   f. Returns to the original calling routine.

It is necessary to make calls in this order, because the data mapping must be done in a PL/I environment.

ASSEMBLER Option

The optimizing compiler provides a facility to simplify calling assembler language routines from PL/I. This consist of setting up an argument list that contains the addresses of all items passed rather than the addresses of locators.

When an entry point is declared as OPTIONS (ASSEMBLER), parameter lists passed to the entry point are set up to contain no locator addresses. The addresses of any areas, arrays, strings, or structures are passed directly in a parameter list. (For a call to a PL/I routine, the parameter list would contain the address of locators for these data types. This is because the called routine might require information on the length or bounds of the data and this is accessible through the locator. See chapter 4 for details.)

The ASSEMBLER option does not provide facilities for automatically overriding PL/I interrupt handling, nor does it allow PL/I routines to be called from assembler language. If the programmer requires these facilities, he must either provide the necessary code himself or use the COBOL option. The COBOL option without the INTER option provides complete facilities for calling, or being called by, assembler routines. However, its use involves the overhead of calls to the PL/I library interlanguage communication routines.

Full instructions on how to use PL/I with assembler language are given in the programmer's guide for this compiler.

COBOL Option in the Environment Attribute

A separate interlanguage communication facility offered by the compiler is the use of the COBOL option in file declarations. This option allows data sets created by COBOL programs to be read by PL/I programs and allows data sets to be created by PL/I programs in a format that is usable by COBOL programs. Interchange of data sets presents no problems, unless structures are used in the data set. If structures are used, their mapping may be different. (See above, under the heading "Handling Data Aggregate Arguments.") When structures are involved and the mapping is not known to be

the same, both COBOL and PL/I structures are mapped, and compiled code transfers the data between structures immediately after reading the data for input, and immediately before writing the data for output.

During compilation, the compiler examines the record variable to see if any structures are involved. If no structures are involved, no further action need be taken. If structures are involved, a test is then made to see if the mapping of the structure or structures will be the same in COBOL and PL/I. If the compiler can determine that the mapping will be the same, then no action is required. If the compiler cannot determine that the mapping will be the same or if the structure is adjustable, both structures will be mapped. Adjustable structures will be mapped during execution by the resident library structure-mapping routines. Other structures will be mapped during compilation.

When re-formatting of data is necessary, the following actions take place when a record I/O statement involving a file with the COBOL option is executed.

INPUT:

The data is read into a structure which has been mapped using the COBOL mapping algorithm and assigned to a PL/I mapped structure.

OUTPUT:

Before the output takes place, the data in the PL/I structure is assigned to a structure mapped for COBOL. The output to the data set then takes place from the second structure.

The data assignment is carried out by compiled code in all circumstances.

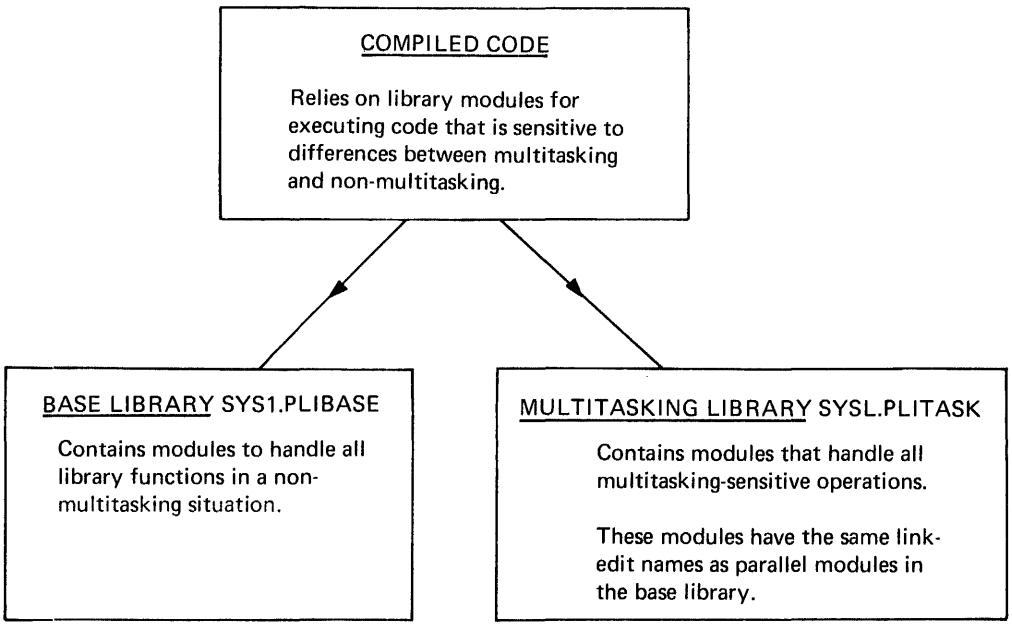


Figure 14.1. Multitasking is implemented by use of a multitasking library

Chapter 14: Multitasking

Introduction

Multitasking allows the PL/I programmer to make use of system multiprogramming facilities within a single jobstep. The PL/I main procedure and certain other PL/I procedures are attached as tasks, and compete for the facilities of the CPU.

All features of the PL/I language that are implemented differently for multitasking and non-multitasking programs are handled by routines in the OS PL/I Resident and Transient Libraries. The non-multitasking routines are held in the partitioned data set SYS1.PLIBASE; the multitasking routines are held in the partitioned data set SYS1.PLITASK. When a multitasking program is link-edited, the automatic call library must be identified by sequential SYSLIB DD statements specifying first SYS1.PLITASK and then SYS1.PLIBASE.

Subroutines that have the same function in both the multitasking and the non-multitasking libraries have the same link-edit name (see chapter 3). Consequently, no special calls are required in compiled code. If the program uses multitasking, the multitasking version of the library module will be link edited, provided that SYS1.PLITASK is specified before SYS1.PLIBASE. Where a module is required only for multitasking programs, it is addressed from the TCA. The results of attempting to access such a module in a non-multitasking program are unpredictable. The concept of the multitasking library is shown in figure 14.1.

The use of a special multitasking library to handle all code that is affected by multitasking minimizes the effect on compiled code. Special action is required only for a CALL statement with any of the multitasking options, and for the epilogue of a block that contains a CALL statement with multitasking options. Otherwise, the code generated for a multitasking program is exactly the same as the code generated for a non-multitasking program. The TASK option on a procedure statement, necessary with some compilers, is ignored by the optimizing compiler.

The Concept of the Control Task

To implement PL/I multitasking, the facilities offered by the operating system control program have to be used in a manner that meets the specifications of the PL/I language. Certain facilities offered by PL/I, notably the ability of any task to change the priority of any other task, are not directly available in the system. Consequently, an interface is used between the system facilities and PL/I tasks. This interface takes the form of a control task.

The control task has all PL/I tasks attached as direct subtasks and always has a higher priority than any PL/I task. Certain functions are always carried out within the control task. These functions are:

1. Attaching and detaching of tasks
2. Accessing or altering COMPLETION or PRIORITY values
3. Modification of event variables (except for STATUS pseudovariable)
4. Generating PL/I dumps
5. Access to IOCBs (see chapter 8) in certain conditions.

The first two are carried by the control task because of the demands of the system control program. The third is carried out by the control task because it is important that no two tasks try to access the event variable chain at the same time.

The apparent and actual hierarchy of tasks is shown in figure 14.2. The functions executed in the control task are shown in figure 14.3.

Throughout most of the execution of a PL/I multitasking program, the control task is in a wait state and the various PL/I tasks are competing for the facilities of the CPU. The control task waits on an ECB list that contains an ECB (event control block) for each PL/I task and an ECB known as the task-end ECB that is used when terminating a task. Whenever any of the functions that must be carried out in the control task are required, the ECB associated with the requesting task is posted with a request code and the task goes into a wait state, waiting on an ECB that is posted complete when the requested

function has been executed in the control task.

Communication between Tasks

As explained above, there is no communication between PL/I tasks except through the control task. Communication between the control task and the PL/I tasks is made through control blocks known as tasking appendages. Every PL/I task has a tasking appendage, which is addressed from and is contiguous with the TCA of the task.

As shown in figure 14.4, every tasking appendage is headed by an ECB, followed by two fullwords for parameters, followed by another ECB.

The first ECB in the tasking appendage is known as the POST ECB, and is one of the ECBs in the ECB list on which the control task waits. The second ECB is known as the WAIT ECB and is the ECB on which the task waits while a function is carried out in the control task.

When code within a subtask requires a service to be done in the control task, it posts the POST ECB with a completion code to identify the service required, and waits on its WAIT ECB. The WAIT ECB will be posted complete when the requested action has been completed in the control task.

The completion codes that are used to post the POST ECB are:

COMPLETION PSEUDOVARIABLE POSTCODE	X'0'
EVENT ASSIGNMENT POSTCODE	X'4'
PRIORITY PSEUDOVARIABLE POSTCODE	X'8'
I/O EVENT COMPLETION POSTCODE	X'C'
WAIT TERMINATION POSTCODE	X'10'
EXECUTE IN CTRL TASK	X'14'
DEDICATE CONTROL TASK ROUTINE	X'18'
LIBERATE CONTROL TASK ROUTINE	X'1C'
ATTACH A TASK	X'20'
END OF TASK	X'24'
TERMINATE SUBTASK	X'28'
TERMINATE SUBTASK	X'2C'

Any parameters required are passed to the control task in the list that follows the POST ECB.

Holding the Priority of the Task

The control program retains the priority of a task in an associated TCB (task control block). At the PL/I level, however, the priority is held in a task variable. This allows the priority of the task to be set even when the task is inactive, and also allows reference to the task by the program. Each task has a task variable which is connected to the TCB through the tasking appendage. The address of the associated tasking appendage is placed in the task variable when the task is attached.

When a change in the priority of a task is requested, the priority is always changed in the task variable. If the task variable is active, the priority is also changed in the TCB.

Also associated with a task is an event variable. The event variable is set "complete" when the task is terminated.

All tasks have associated event and task variables. If none are specified by the programmer, dummy variables are provided during task attachment. These dummies are held in the task's own workspace, and are discarded when the task is terminated.

Multitasking Housekeeping

Multitasking housekeeping is similar to non-multitasking housekeeping. Every task has its own TCA and other blocks in the program management area, as described in chapter 5.

The major differences are that the TCA for each task has a control block known as the tasking appendage, and that DSA chaining between tasks cannot follow the rules of calling procedures.

As shown in figure 14.6, the chaining of DSAs is arranged so that the dummy DSA of the attached task is in the chain but the DSA of the attaching procedure is not. This protects the attached tasks from any changes in establishment of on-units that may occur in the block that attached the task. In order that error handling and other functions using the backchain may

PL/I PROGRAM

```
X:PROC;
.
.
.
CALL Y TASK (T1) EVENT (E1);
.
   Y:PROC;

   CALL Z TASK (T2) EVENT (E2);
     Z:PROC;

     END Z;
   END Y;
END X;
```

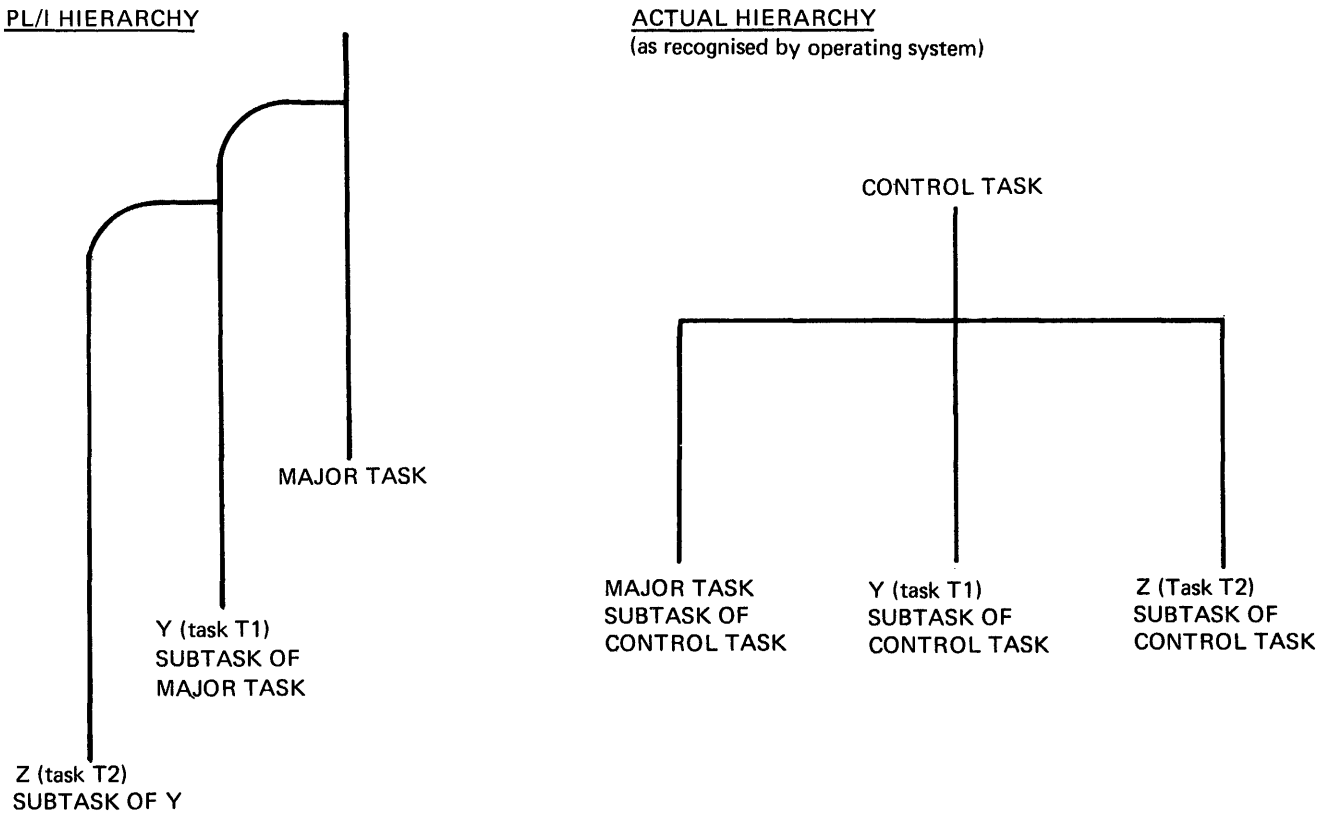


Figure 14.2. The hierarchy of tasks

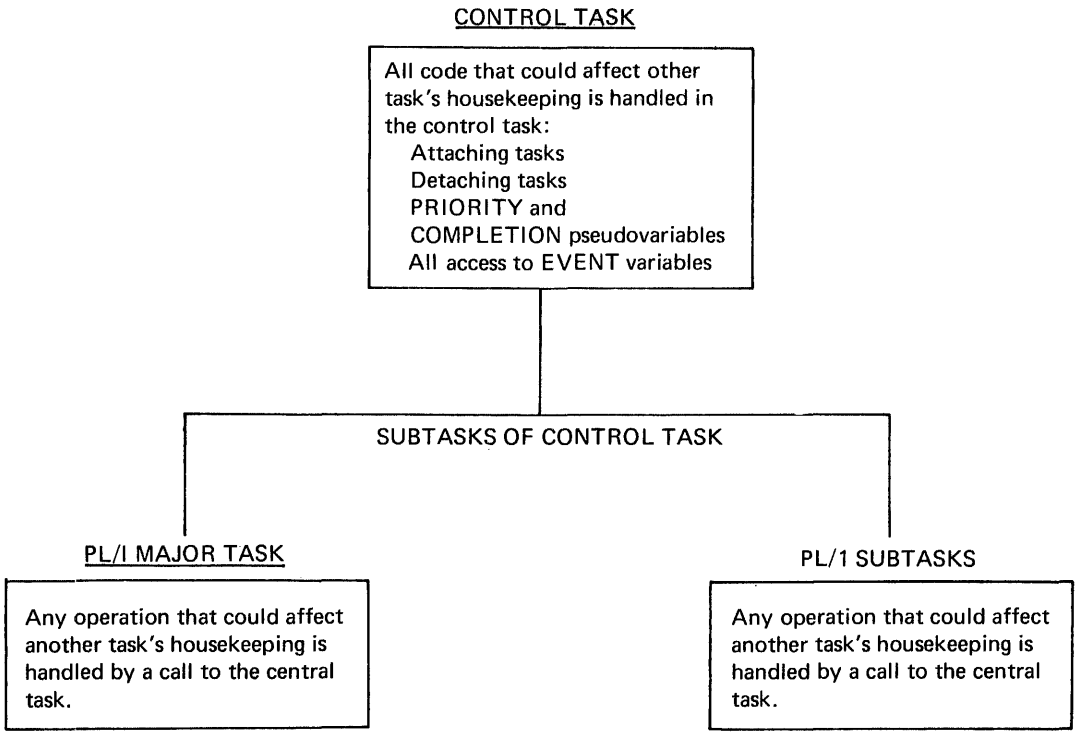


Figure 14.3. The functions of the control task

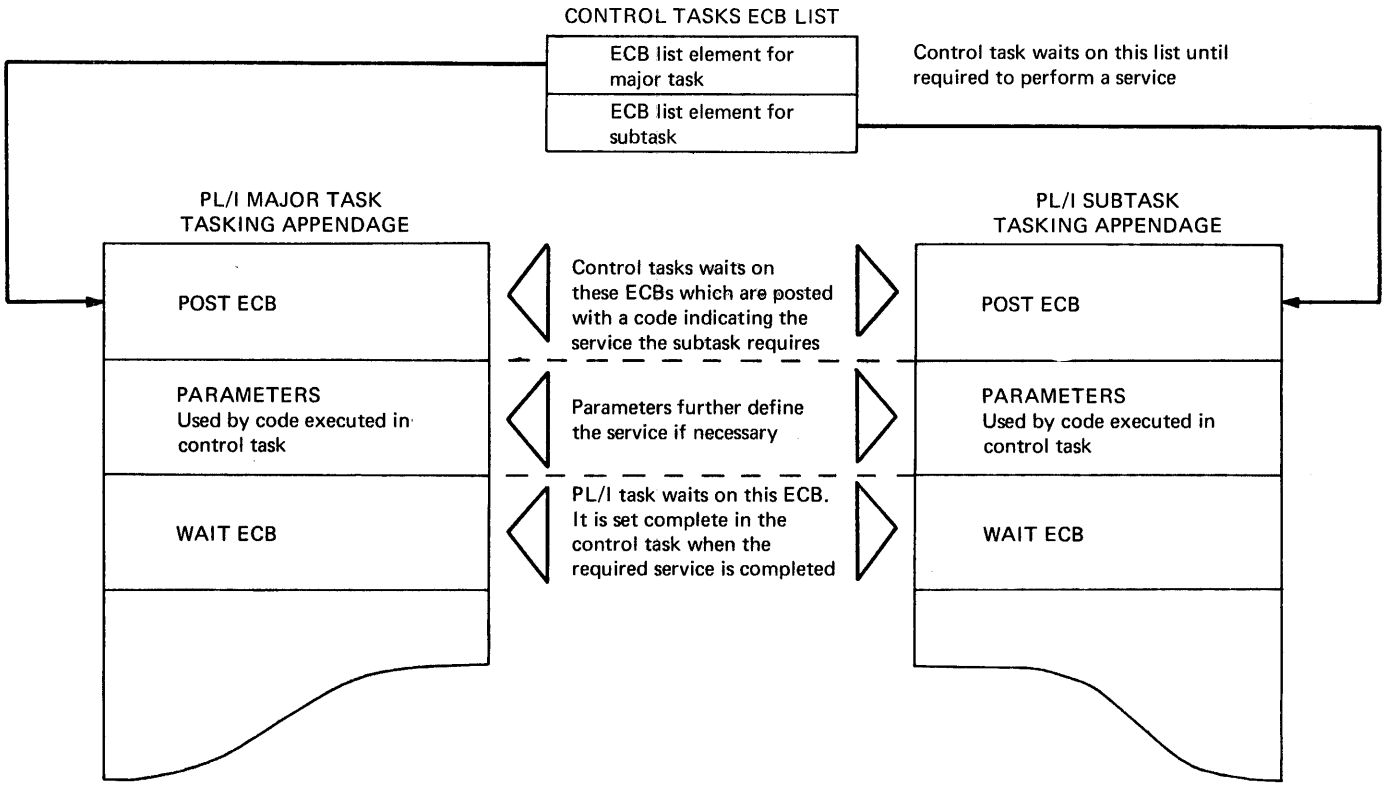


Figure 14.4. The post and wait ECBs

Modules in the tasking library		
Control Name*	Link-edit Name*	Function
IBMTPIR	IBMBPIRA	Program initialization and task housekeeping
IBMTPGR	IBMBGGRA	Storage management
IBMTPGO	IBMBPGOA	Abnormal GOTO
IBMTTOC	IBMBTOCA	COMPLETION pseudovariable
IBMTTPR	IBMBTPRA	PRIORITY pseudovariable
IBMTJWT	IBMBJWTA	WAIT statement
Multitasking Modules in the Transient library		
IBMTPJR	IBMTPJRA	Program initialization and task housekeeping
IBMTPJI	IBMTPJIA	Program initialization

* Control name is the name that uniquely defines the module.

Link-edit name is the name by which a module is known to the linkage editor. Multitasking and non-multitasking modules that handle similar functions have the same link-edit name.

Figure 14.5. Modules in the multitasking library

function correctly, certain items, such as on-cells and dynamic ONCBs, are copied from the attaching task's DSA to the dummy DSA of the attached task at the time of attachment.

If procedures executed as separate tasks are internal to one another, a static backchain is established through the DSAs. This backchain passes from the attached task's procedure DSA to the DSA of the procedure in which the task was attached, and is the same as for non-multitasking programs. This chaining allows all internal procedures to access variables declared in outer blocks without requiring special provision for multitasking. (Special action is, however, necessary when handling the CHECK condition.)

To maintain the PL/I hierarchy, more information than is available in the DSA chain is required. In addition to the DSA chain, tasks with the same attaching task are chained together, and the most recently attached subtask is chained to its parent task. The chains between tasks with the same attaching task are known as sister task chains. The sister task chains and the chain to the most recently attached subtask are all held within the tasking appendage. The chaining arrangement, shown in figure 14.7, allows quick access to all related tasks.

The sister task chain goes in both directions. Each task is chained to the task attached immediately before it (elder sister) and the task attached immediately after it (younger sister). The most recently attached task has no younger sister. Its younger sister chain points instead to the attaching task. However, instead of pointing at the head of the tasking appendage, it points at offset X'8' within the tasking appendage. The effect of this is that an attempt to continue to follow the younger sister chain results, beyond the attaching task, in access not to the younger sister pointer but to a field offset from it by X'8'. This field, which is always set to zero in all tasks, is known as the stopper field. Access to it indicates that the attaching task has been reached.

When a task is terminated, all its subtasks must be terminated. To simplify finding these tasks, a flag is set in the DSA of the block in which a task is attached. The flag remains set while any active tasks are attached.

The Multitasking Library

Module IBMTPIR loads IBMTPJR to perform most multitasking functions. IBMTPJR carries out the majority of functions that are executed in the control task. IBMTPJR issues a LOAD macro instruction to pass control to IBMTPJI to perform parameter translation, and to initialize the control task and the storage for the major task. IBMTPJR then attaches the major task. IBMTPJR also contains the instructions to handle the major functions which have to be carried out within the control task. Each of these functions is handled by a particular subroutine within IBMTPJR. A simplified flowchart of IBMTPJR is shown in figure 14.8.

The program initialization module IBMTPJR has a register save area, but is unlike other PL/I library routines in not having a DSA. IBMTPIR acquires workspace,

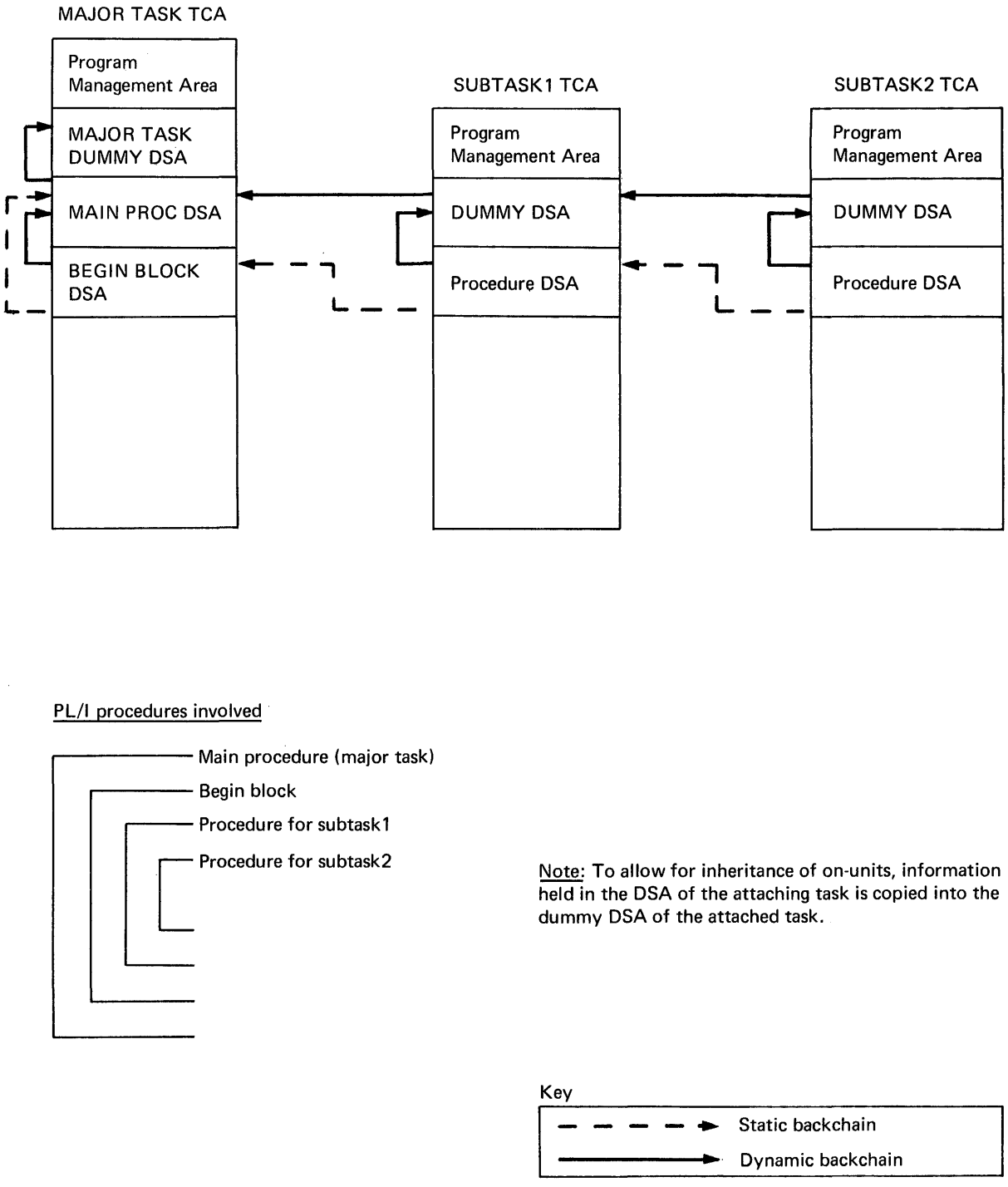
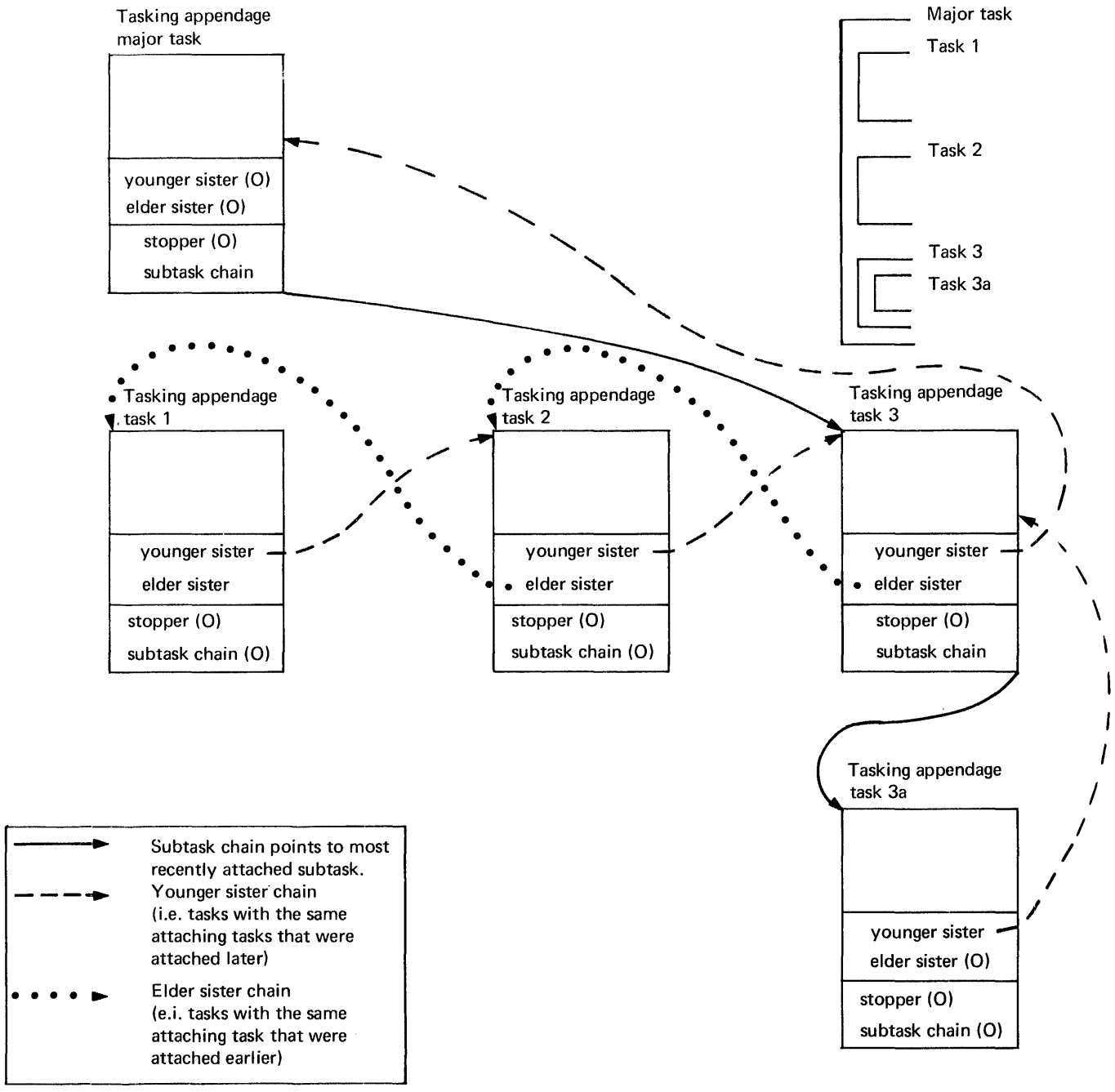


Figure 14.6. Backchains in multitasking



Note: Because tasks are chained in both directions, all relationships be quickly found.
Following the 'younger sister chain' leads to the attaching task. When the attaching task is reached, the offset that should be the offset to the younger sister is to the stopper. Thus it is known that the attaching task has been reached.

Figure 14.7. The chaining of tasks through their tasking appendages

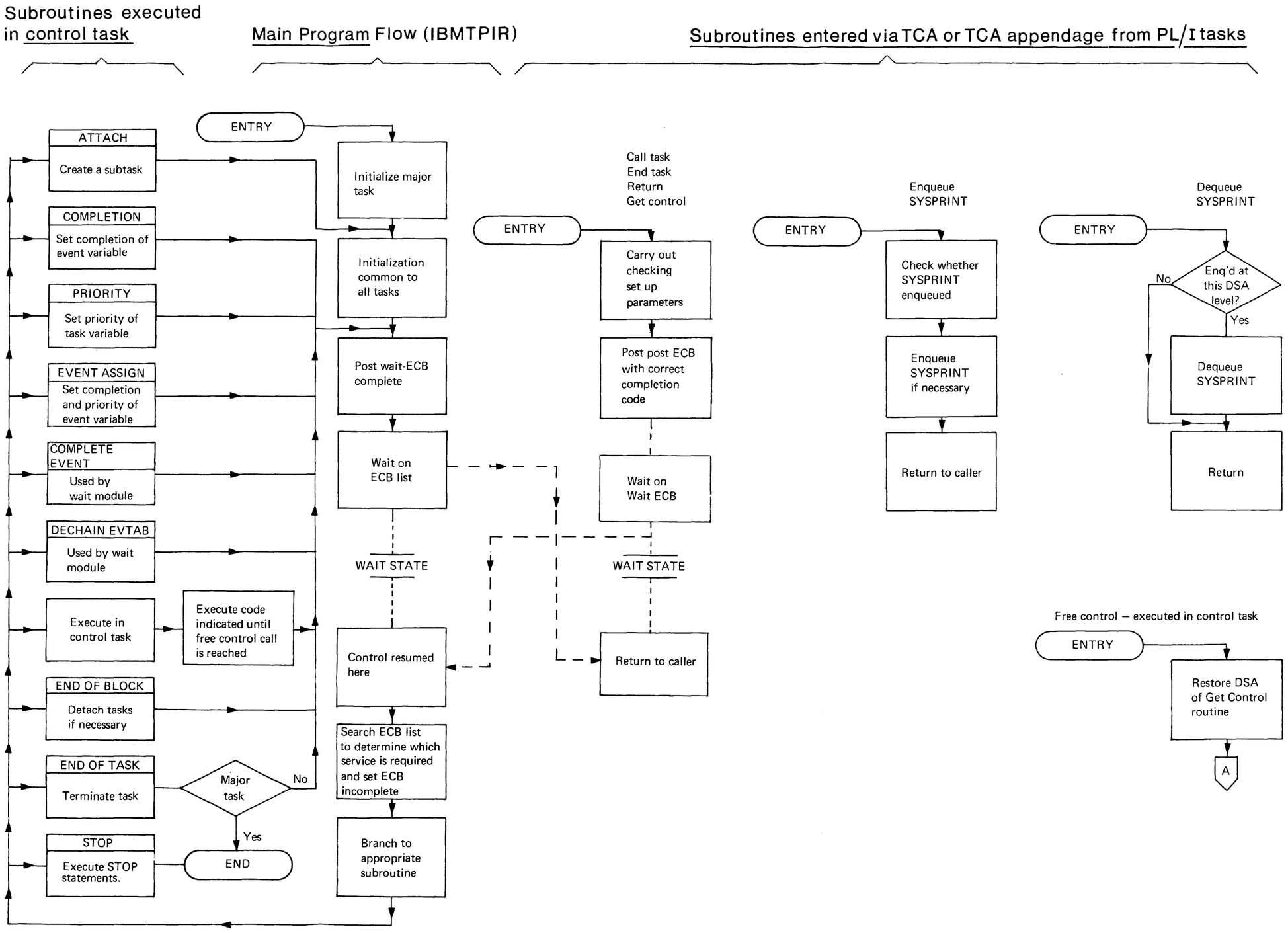


Figure 14.8. A simplified flowchart of IBMTPIR

contiguous with the standard register save area, to hold: the addresses of the ECB lists; the address of the area where the next ECB-list element will be placed; the task-end ECB (used when detaching a task - see below), the diagnostic file block, and the dump block. These last two blocks are held in the control task workspace because they must serve for all PL/I tasks.

Supporting IBMTPJR are two routines that are link-edited only when necessary: IBMTTOC is link-edited only if the COMPLETION pseudovariable is used; IBMTPRA is link-edited only if the PRIORITY pseudovariable is used.

Also included in the multitasking library are a number of routines that handle action which requires different machine instructions for a multitasking program. Among these routines are storage management and error handling routines.

All the routines in the multitasking library are shown in figure 14.5 and described in OS PL/I Resident Library: Program Logic.

How the Control Task Operates

The control task is created by the system when the PL/I program is initialized. The instructions first executed within the control task are in the program initialization routine IBMTPIR. This routine is entered because its address is specified in the control section PLISTART (see chapter 5).

IBMTPJR obtains a standard save area, and then loads and branches to IBMTPJR which performs the remainder of the initialization.

IBMTPIR sets up the environment for the major task, which it then attaches with an ATTACH macro instruction, after further initialization, control is given to the address held in PLIMAIN

IBMTPJR then builds an ECB list which consists of the WAIT ECBs for the PL/I task that has been attached plus the task-end ECB. A wait is then issued on this ECB list, and the control task will remain in the wait state until the major task requires a service that must be handled in the control task.

When control returns to the control task, execution recommences in IBMTPJR immediately after the point at which the WAIT macro instruction was issued. The action at this point is to search the ECB list, discover which ECB has been posted, and then to carry out the action specified in the code posted in this ECB. The action is carried out by calling a subroutine of IBMTPJR. This subroutine may perform the function required, execute a sequence of requested instructions, or call further library routines to handle the requested function.

Whenever a new subtask is attached, a further POST ECB is added to the ECB list of the control task.

Whenever PL/I tasks require a service that is handled in the control task, a call is made to a library entry point. The majority of calls are to subroutines of IBMTPJR, which are addressed via the TCA or the TCA appendage. However, the PRIORITY and COMPLETION pseudovariable routines are separate library modules. This saves space in programs where the pseudovariables are not used.

Attaching a Task

A CALL statement with one of the multitasking options is compiled as a call to an entry point in IBMTPJR. This entry point is addressed via a module list whose address is held in the TCA. The entry point is passed the address of the procedure that is to be executed as the attached task, and any parameters that are to be passed to that procedure.

The routine in IBMTPJR posts the POST ECB for the attaching task with a completion code of 24, indicating that a new task is to be attached. It then issues a WAIT macro instruction on its own WAIT ECB, and the attaching task goes into the wait state.

Control passes to the control task. The first action of the code within the control task is to scan the ECB list to see which task is requesting a service, and which service is being requested. According to the completion code in the ECB, one of the subroutines in IBMTPJR is entered. For attaching a task, the attach-task subroutine is entered. The minimum storage the subroutine attempts to acquire is a new program management area. Depending on the options in the ISASIZE parameter, it may also attempt to acquire storage for DSAs and other dynamic requirements.

The new program management area is set up within the storage acquired, and the new TCA is placed at the head of the chain of daughter tasks that is held in the attaching task's TCA.

The new TCA is then associated with a task variable and an event variable. If these were specified in the CALL statement, they are used. Otherwise, dummy event and task variables are set up by IBMTPJR. These dummy variables are held in the working storage of the new block. The event and task variables are then chained to and from the TCA. A bit is set in the DSA of the block that was being executed when the task was attached.

The PRV of the attaching task is then copied into the attached task. This ensures that addressing information for files and controlled variables cannot be altered by the attaching task. Similarly, on-unit establishment information is copied from the attaching task's current DSA into the dummy DSA of the attached task. This ensures that the subtask acts according to the situation prevailing at the time when the call was made.

The attaching routine finally sets the POST ECB of the new task incomplete, adds this new POST ECB to the control task's ECB list, completes the ECB on which the requesting task is waiting, and issues a WAIT macro instruction on the control task's ECB list.

The newly attached task and the original requesting task are now both ready to receive control from the control program. The control task is in a wait state, ready to service any further requests from PL/I tasks.

Failure of CALL...TASK Statements

A number of situations can cause a CALL..TASK statement to fail. These situations are:

1. Too many tasks are already active
2. There is insufficient storage for the new task
3. The task variable is already active
4. The event variable is already active

In any of these situations, the calling task is posted with a non-zero postcode. When this postcode is detected, the task generates the correct error code, and calls the error handler.

Detaching a Task

Tasks are normally detached when they reach any EXIT statement, or an END or RETURN statement in the procedure that was attached as a task. In such circumstances, control returns in the normal manner to IBMTPIR, whose registers have been stored in the dummy DSA of the task. IBMTPIR is then in a position to pass control to the control task, so that the requesting task can be terminated. After housekeeping operations, the control task sets the priority of the task to be detached as high as possible, completes the WAIT ECB of the task, and then waits on the task-end ECB. When the task to be terminated resumes control, it posts the task-end ECB complete, and terminates itself by returning to the control program.

The process described above is used because it is simpler than handling the ABEND that would otherwise result when one task is detatched from another.

Abnormal Termination of a Task

When a block is terminated, any tasks attached during the execution of the block are also terminated. For this reason, epilogue code of blocks in which tasks may be attached contains a call to a subroutine of IBMTPIR. This subroutine passes control to the control task, from which the purge task subroutine is called. This routine examines the DSA of the block being freed, to see whether any active subtasks remain; if any do remain, they are terminated.

Active subtasks are accessed via the chain of daughter tasks from the TCA of the task in which the block is being terminated.

Abnormal termination of a task involves ensuring that any WAIT statements being executed by the task are properly terminated, event variables are completed, task variables are set inactive, and ECB elements are removed. Event I/O operations started in the tasks are completed.

The Get-Control and Free-Control Routines

In order to increase the scope of jobs that can be handled within the control task, the program initialization routine includes a facility whereby a request can be made for any defined sequence of instructions to be

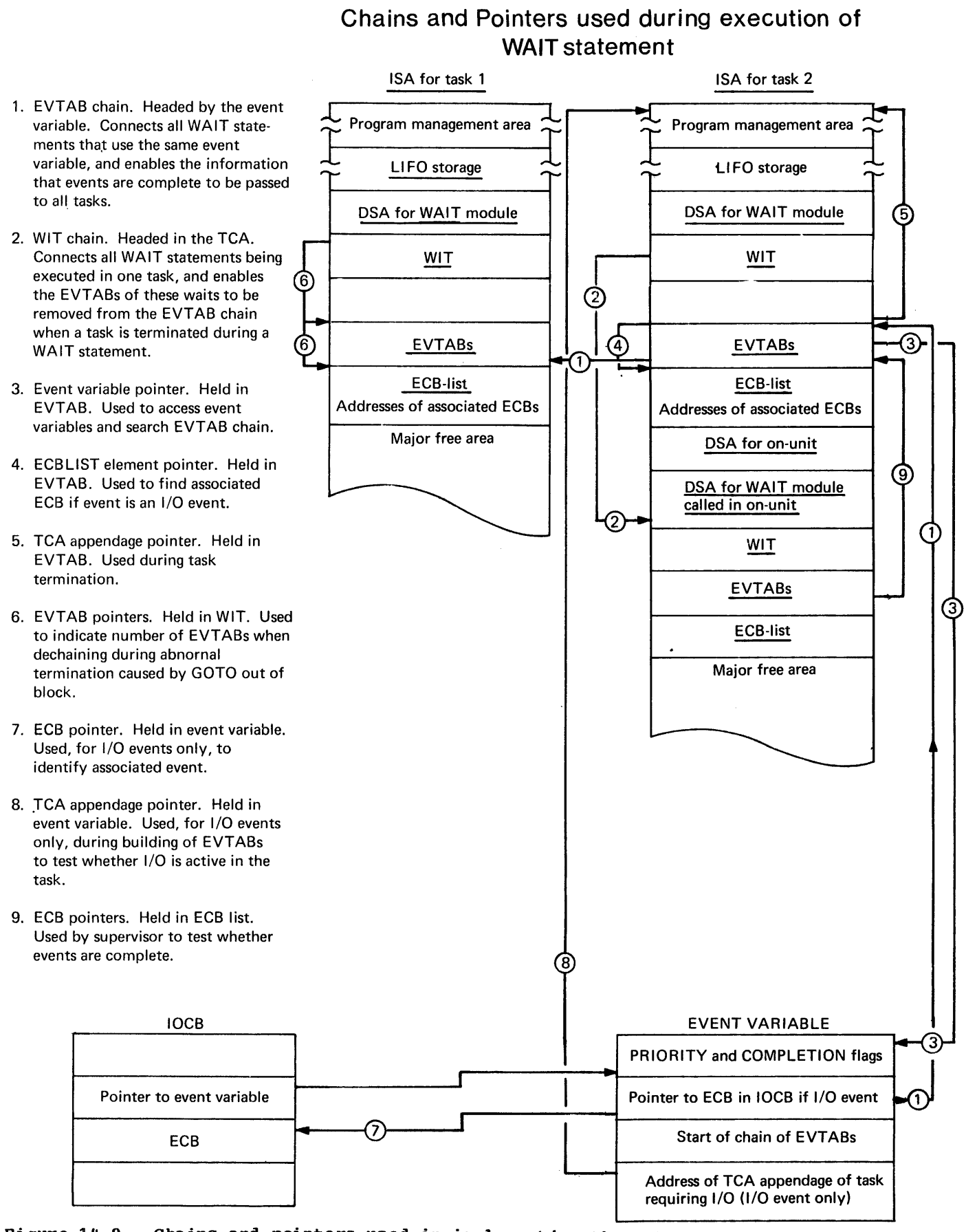


Figure 14.9. Chains and pointers used in implementing the WAIT statement

executed within the control task. This facility is used by a number of library routines when accessing event variables, or carrying out other actions that have to be executed within the control task. It is not used by compiled code.

The instructions to be executed within the control task are delimited by calls to two library subroutines, whose addresses are held in the TCA. These routines are the get-control and free-control routines. Both are subroutines of IBMTPIR.

When the get-control routine is called within a PL/I task, it saves the caller's registers, posts its POST ECB, and issues a wait on the requesting task's ECB.

When the control task gains control, it restores the registers saved by the get-control routine, and branches to the address in register 14. The address will be the instruction after the call to the get-control routine, because the routine was called in the standard manner, that is, a BALR instruction on registers 14 and 15.

Execution of the instructions then continues in the control task until a call to the free-control routine is met. This routine stores the current registers in the DSA of the block that originally called the get-control routine. The free-control routine now posts the WAIT ECB of the requesting task, and resets the control task waiting on its ECB list.

During execution of the free-control routine, the routine modifies the value in the register 14 save area in the DSA of the block that originally called the get-control routine. When control returns to the original requesting task, it returns to the point in the get-control routine immediately following the point where the WAIT was issued. The get-control routine restores the register values, and branches to the new address in register 14.

The required instructions have now been executed within the control task, and execution can continue in the original task. The processes involved in the get-control and free-control routines can be followed in the flowchart of IBMTPIR in figure 14.8.

Altering COMPLETION and PRIORITY Values

To prevent two PL/I tasks attempting to alter the completion and priority values of tasks or events at the same time, alteration of these values is always done by code in the control task.

When such access is required, compiled code in the requesting task branches to a library subroutine that posts the control task with a completion code in the POST ECB, and issues a wait in the requesting task. When the control task receives control, it inspects the completion code, and calls a subroutine in IBMTPIR. For the PRIORITY pseudovariable, the subroutine in IBMTPIR calls a subroutine in IBMTTPR to handle the actual alteration. This is to save space in programs where the PRIORITY pseudovariable is not used.

The subroutine accesses and alters the values as requested. Where necessary, a CHAP macro instruction is issued to alter the priority of a task.

Executing the WAIT Statement

The WAIT statement can be used in both multitasking and non-multitasking programs. A description of WAIT in the non-multitasking situation is given in chapter 11.

At the PL/I level, each WAIT statement is associated with one or more events, and each event is associated with an event variable. When the specified number of these event variables is set "complete," the wait is terminated.

PL/I event variables are not accessed by system wait macro instructions; they contain a pointer to the event's ECB. This ECB will have been nominated in the WAIT macro instruction issued to the system, and will be set complete when the associated event is complete. When the event is complete, the PL/I program can inspect the ECB, and complete the event variable.

The PL/I event variable cannot be used to indicate to all WAIT statements nominating the associated event that the event is complete. This is because an event variable may be associated with a further event immediately after completion of the event with which it was formerly associated. If more than one task is waiting, this may be before all the WAIT statements nominating the event are satisfied. See figure 14.10.

To overcome this problem, a control block known as an EVTAB is used. An EVTAB is generated for every WAIT statement. For every event nominated in the statement, an EVTAB element is produced, containing the ECB for the event and a pointer to other EVTAB elements associated with the event. Thus, when an event is completed in one task, the chain from the event variable is

scanned and any ECBs associated with the event are set complete.

A further control block is used in the implementation of the WAIT statement. This is the wait information table (WIT). A WIT contains a record of any WAIT statements that are being executed in a particular task. This information is used when a task is being terminated, because any active events must be removed from the chain that associates event variables with EVTABs. Were this not done, the chaining of EVTABs would be destroyed because the EVTABs in the terminated task would be lost.

The chaining of the control blocks described above is shown in figure 14.9.

The Wait Module IBMTJWT

The WAIT statement is executed by means of a call to the wait module, IBMTJWT. The module is passed a list of event variables and, optionally, a value indicating how many of the events must be completed before the wait is satisfied. If no value is specified, all events must be completed.

The wait module may be passed various types of event variable:

1. Active event variables. These are associated with:
   a. I/O or display events that were initiated in the current task.
   b. I/O or display events that were initiated in another task.
   c. Events associated with tasks.
2. Inactive event variables. These are associated with events that must be completed by use of the COMPLETION pseudovariable.
3. Incompletable event variables. These are associated with events that have caused entry to an on-unit because an I/O condition has been raised in the current task, and which cannot be completed because the on-unit also specifies a wait on the event that is already being waited on.

If any of the events are incompletable, IBMTJWT checks to see whether the WAIT statement can be satisfied by completable events. If the WAIT statement cannot be satisfied, an attempt is made to complete all I/O and display events initiated in the current task, as other tasks may be waiting on these events. When these events are completed, and the associated ECBs in other tasks set complete, the error handler is called to terminate the current task.

If the WAIT statement can be satisfied by completable event(s), the incompletable event is ignored.

If any of the events are I/O or display events initiated in the current task, an ECB will already have been created for these events when the statement with the EVENT option was executed. This ECB must be accessed and waited on. Access is made through the event variable.

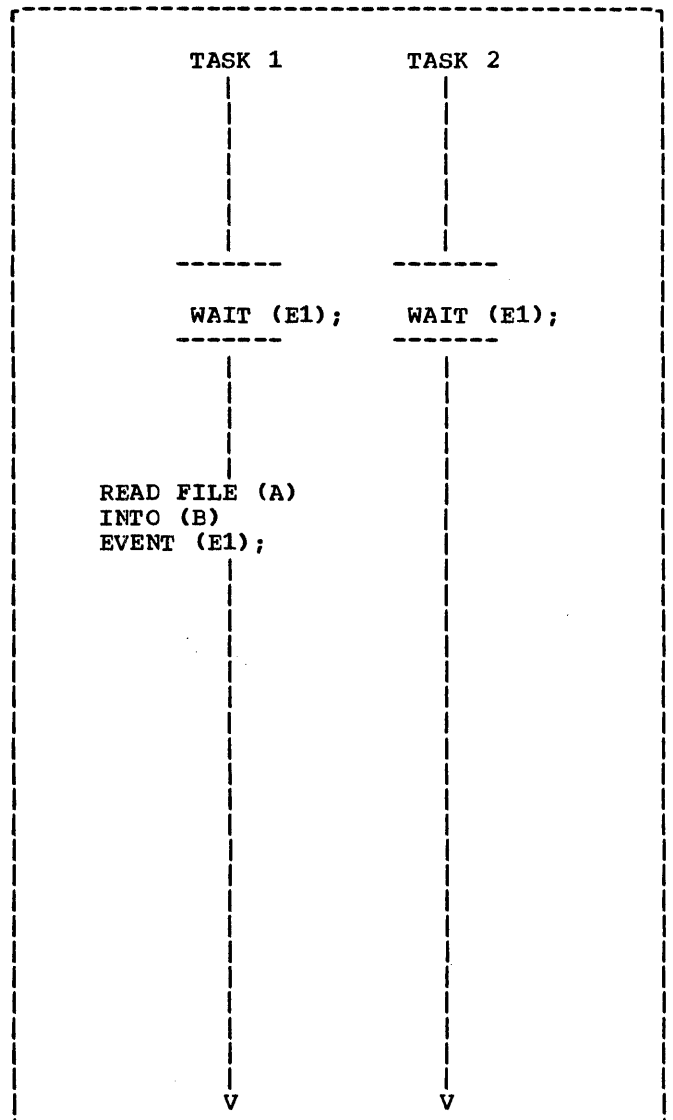
Note that for I/O events, a CHECK macro instruction is issued by the I/O transmitter. If all events are I/O events initiated in the current task, and all of them have to be completed, it is possible to use the CHECK macro instruction to satisfy the WAIT statement. The wait module passes the events one at a time to IBMBRIO. Return is made when the event is complete. The wait module then searches the EVTAB chain, setting any associated ECBs complete. It then passes the next event to IBMBRIO, continuing the process until all events are complete. If all events need not be completed, this method cannot be used, because one of the events nominated might prove incompletable and, consequently, the task would be terminated.

If the events are not I/O or display events initiated in the current task, the wait module builds an EVTAB element for the event, and associates it with the event variable. If only one event is involved, the wait module then issues a WAIT macro on the ECB; if more than one event is involved, the wait module places the address of the ECB in an ECB list on which a WAIT macro instruction will be issued.

If the wait module issues a WAIT macro instruction on an ECB list, control will return to the module when one or more of the ECBs has been completed.

The wait module scans the EVTAB elements and discovers which of the events has been completed. If the event is an I/O event in the current task, it will be necessary to complete the event variable and scan the EVTAB chain, completing ECBs in any tasks that are waiting on the event that has been completed. The ECBs are completed by calling a subroutine of IBMTPIR, which executes the necessary instructions in the control task. The subroutine completes the ECBs by means of a POST macro instruction.

If the wait is to be made on events that can only be completed in other tasks, the wait module issues a WAIT macro instruction specifying that all the events in the ECB



Task 1 reuses the event variable E1. If task 1 acquires control before task 2 on completion of the original event with which E1 is associated, then the event variable on which task 2 is waiting will be associated with an event other than that originally intended. A mechanism to supplement the event variable is therefore needed: this mechanism is known as the EVTAB chain. In the above example, the EVTAB chain would allow task 2 to determine that the original event was complete.

Figure 14.10. Reusing event variables, and the need for the EVTAB chain

list must be completed.

When all completed ECBs have been handled, the ECB list and the EVTAB elements are rebuilt for all events that are not complete. A further WAIT macro instruction is issued on the ECB list, and the process is continued until the necessary number of events have been completed.

If the number of events needed to satisfy the WAIT statement are complete, but further events remain incomplete, it is necessary to dechain EVTABs from the chains associated with the incomplete events. This is done by a call to a subroutine in IBMTPIR, which executes instructions in the control task to remove unneeded EVTAB elements from the EVTAB chain.

If the WAIT statement specifies only active events, no further action can be taken until the events are complete. Accordingly, the wait module issues a WAIT macro instruction specifying that all events have to be completed. Thus control will not return to the task until the wait is satisfied.

Enqueuing and Dequeuing on SYSPRINT

In order to protect error messages from interruption by other output to SYSPRINT, or from error messages in different tasks, the error message modules and all calls to SYSPRINT are enqueued and dequeued by means of a call to a subroutine in IBMTPIR, which issues the ENQ and DEQ macro instructions. A call is made immediately before and immediately after the output.

Similar action is taken on EXCLUSIVE files, for which the ENQ and DEQ macro instructions are issued by the library module IBMBPQD.

Appendix A: Control Blocks

This appendix provides information on the format of the control blocks that may be used during the execution of a program compiled by the OS PL/I Optimizing Compiler. Brief details of the function of each control block, together with when it is generated and where it can be located, are also given.

Except where explicitly stated all offsets from the start of a block are byte offsets and are given in hexadecimal notation.

Area Locator/Descriptor

Function

Holds the address and length of the area variable for passing to other routines or for execution time reference if the area has an adjustable length.

When Generated

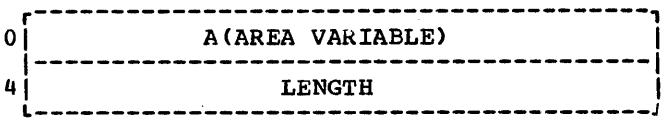
As far as possible during compilation. If necessary completed during execution.

Where Held

Static internal control section.

How Addressed

From an offset from register 3 known to compiled code



A(Area Variable) is the address of the area variable control block.

Length is the total length including both the control block and the area variable.

AREA DESCRIPTOR

The area descriptor is the second word of the area locator/descriptor. It is used in structure descriptors, when areas appear in structures, and in the controlled variable 'description' field when an area is controlled.

Area Variable Control Block

Function

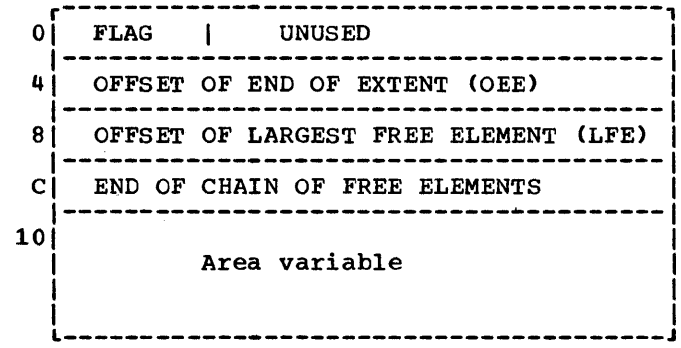
Used to control storage allocation within the area variable.

When Generated

When the area variable is initialized. This depends on the storage class of the area.

Where Held

At the head of the area variable.



Free elements: If there are free elements in the area variable, they are headed by two words. The first word gives the length of the element, the second word gives the offset to the next smaller free element. If there is no smaller free element, the second word is set to zero.

Flag X'0' Area variable does not contain free elements.
X'1' Area variable does contain free elements.

Aggregate Descriptor Descriptor

Function

Contains information needed to map a structure or an array of structures during execution. Used for structures that contain adjustable extents or the REFER option. See chapter 4.

When Generated

As far as possible during compilation. Adjustable values are filled in during execution.

Where Held

Static internal control section.

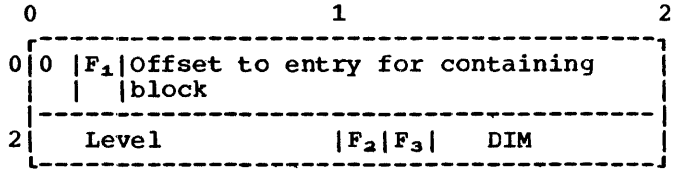
How Addressed

From an offset from register 3 known to compiled code

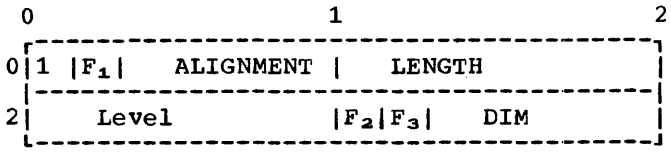
General Format

An aggregate descriptor descriptor consists of a series of fullword fields one for each structure element and one for each base element in the structure.

Structure Element



Base Element



where,

F_1 = '0'B Not last element in structure

= '1'B Last element in structure

F_2 = '0'B Not an AREA

= '1'B An AREA

F_3 = '0'B Not a BIT string

= '1'B BIT string

OFFSET = The offset within the aggregate descriptor descriptor to the entry for the containing structure. The offset is held in multiples of four bytes.

LEVEL = Logical level of identifier in structure

DIM = Real dimensionality of identifier

ALIGNMENT = Alignment stringency

Value(dec.)	Meaning
0	bit
7	byte
15	half-word
31	word
63	double-word

LENGTH = Length (in bytes) of data

LENGTH = 0 for strings and AREAs, whose length is held in descriptors

Aggregate Locator

Function

Used to pass the address of an array or structure and its associated descriptor to a called routine. Also to associate the aggregate with its descriptor during execution.

When Generated

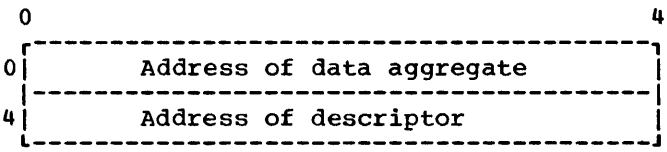
During compilation.

Where Held

Static internal control section.

How Addressed

From an offset from register 3 known to compiled code



Array Descriptor

Function

Contains information about the extent of an array. For arrays of area variables or strings, an area or string descriptor is attached to the array descriptor.

The array descriptor is used to pass information about an array to called routines, or to hold information about an array with adjustable extents.

When Generated

As far as possible during compilation. If the array has adjustable extents, it is completed during execution when the values are known.

Arrays of structures make use of structure descriptors to hold similar information.

Where Held

Static internal control section.

How Addressed

From an offset from register 3 known to compiler code

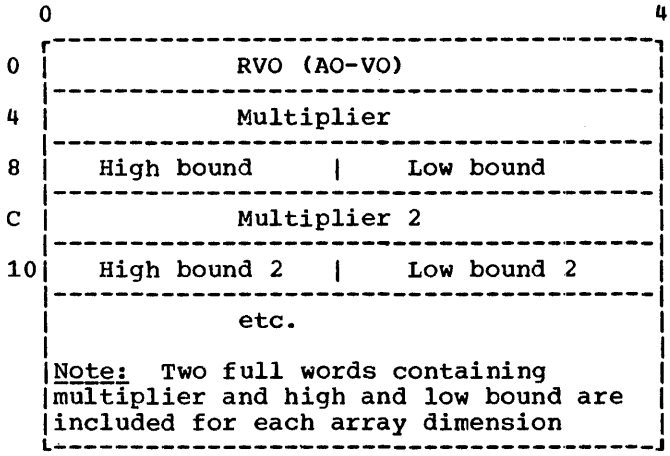
Arrays of Strings or Areas

For arrays of strings or areas, the descriptors are completed by string or area descriptors concatenated to the array descriptor. String and area descriptors are the second word of string and area descriptor/locator pairs.

For bit string arrays, the bit offset from the byte address is held in the string descriptor.

General Format

The first word in the array descriptor is the RVO (relative virtual origin). This is followed by two words for each dimension of the array, containing the multiplier and high and low bound for each dimension.



RVO = Relative virtual origin, the distance between the virtual origin (VO) and the actual origin (AO). Virtual origin is the point at which the element in the array whose subscripts are all zeros is, or would be, held. Actual origin is the start of the first element in the array.

RVO is held as a bit value for arrays of unaligned bit strings, but otherwise as a byte value. Bit offsets are given in the string descriptor. Actual origin and virtual origin are also held as byte values.

High bound:
The highest subscript in any dimension.

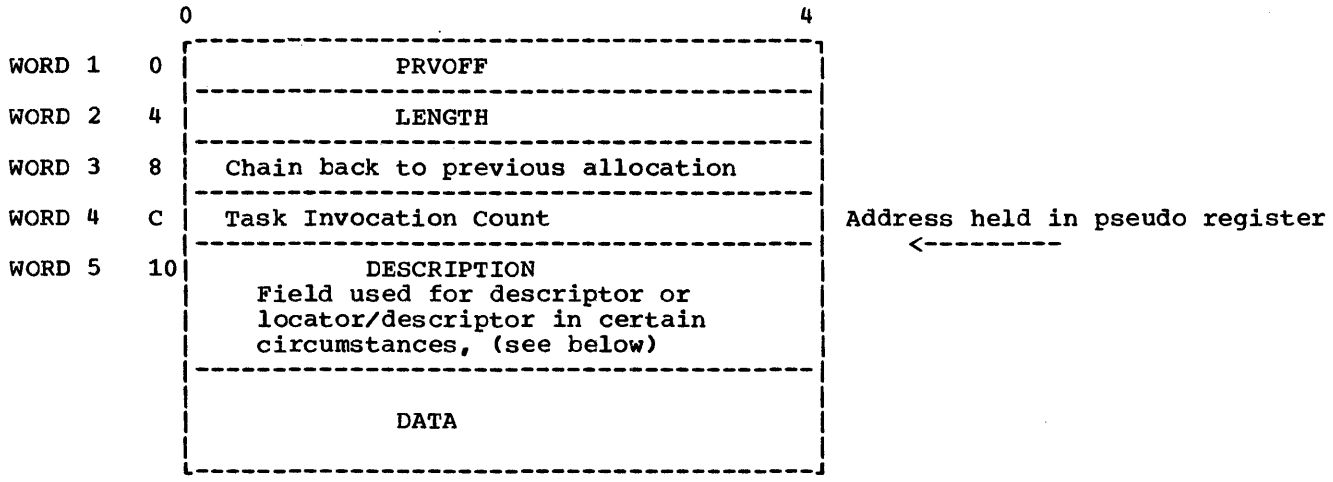
Low bound:
The lowest subscript in any dimension.

Multiplier:
The multiplier is the offset between any two elements marked by the change of subscript number in any dimension.

For example for the array DATA(10,10), the multiplier for the first dimension is the offset between DATA(1,1) and DATA(2,1) etc. The multiplier for the second dimension is the offset between DATA(1,1) and DATA(1,2). The offset is measured from the start of the one element to the start of the next.

Multipliers are byte values except for bit string arrays, in which case they are bit values.

Controlled Variable Block



Function

To hold information about the controlled variable.

When Generated

When the variable is allocated.

Where Held

At the head of the controlled variable.

How Addressed

From an offset in the PRV. (The PRV address is held out offset X'4' in the TCA.)

PRVOFF: Offset within pseudo-register vector associated with the controlled variable.

LENGTH: Length of the total allocation including the 4 words of the heading.

CHAIN BACK: Address of word 5 of previous allocation, set to address of dummy FCB if first allocation.

TASK INVOCATION COUNT: A method of identifying which task the controlled variable is attached to. A controlled variable cannot be freed within a task unless the task invocation count of the variable is the same as that in the TCA.

*DESCRIPTION

If the item is one that requires a descriptor/locator or a locator, this is placed at the head of the data. If the item is a structure or array and the extents are unknown at compile time, the descriptor will also be placed before the data.

Thus for:

STRINGS and AREAS, the controlled variable is headed by a locator/descriptor

STRUCTURES and ARRAYS, the controlled variable is headed by a locator

STRUCTURES and ARRAYS with ADJUSTABLE EXTENTS, the controlled variable is headed by a locator followed by a descriptor

ALL OTHER DATA, the description field is not used and the data itself starts at offset X'10' (16)

Data Element Descriptor (DED)

Function

Used to pass description of data elements to library conversion and stream I/O routines.

When Generated

During compilation.

Where Held

Static internal control section.

How Addressed

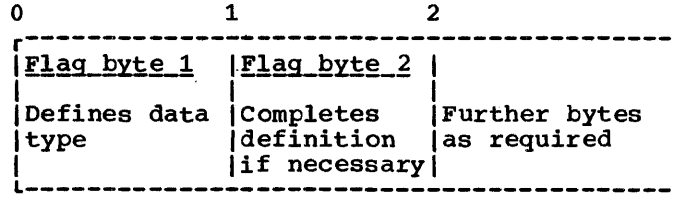
From an offset from register 3 known to compiled code.

Format of DEDs

All DEDs are headed by two bytes that indicate the data type. These two bytes are followed by as many bytes as are required to complete the description of the data.

For arithmetic items, DEDs are completed by such items as scale and precision. For pictured items, a representation of the picture is included in internal form.

General Format



Flag Byte 1 (also known as Code Byte and Look up Byte)

Hex Value	Data Type
00	FIXED BINARY
04	FIXED DECIMAL
08	FLOAT
0C	Free decimal (an internal form)
10	FIXED PICTURE BINARY
14	FIXED PICTURE DECIMAL
18	FLOAT PICTURE BINARY
1C	FLOAT PICTURE DECIMAL
20	non-VARYING CHARACTER
24	non-VARYING BIT
28	VARYING CHARACTER
2C	VARYING BIT
30	CHARACTER PICTURE
40	BINARY constant
44	DECIMAL constant
48	BIT constant
50	F/E Format
54	P Format (arithmetic)
58	A/B/P Format (character)
5C	C Format
60	X Format
64	COL Format
68	SKIP Format
6C	LINE Format
70	PAGE Format
80	LABEL
84	ENTRY
88	AREA
8C	TASK
90	OFFSET
94	POINTER
98	FILE
9C	EVENT

Flag Byte 2

Bits 0&1 = '00'B A-format item
'01'B B-format item
'10'B character picture format item

Bit 2 = '0'B fixed constant
'1'B float constant

Bit 3 = '0'B not extended float
'1'B extended float

Bit 4 = '0'B F-format/fixed picture
'1'B E-format/float picture

Bit 5 = '0'B declared binary
'1'B declared decimal

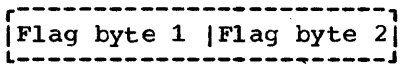
If bits 4 and 5 = '11'B then DED is for character

Bit 6 = '0'B short precision
'1'B long precision

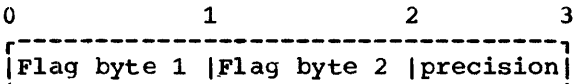
Bit 7 = '0'B real or length specified (A or B format) unaligned bit string
'1'B complex (also set if E, F, or P in C-format) or no length specified (A or B format) or aligned bit string

All bits for which neither value is defined are set to '0'B

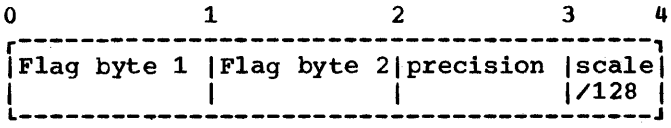
DED for STRING data



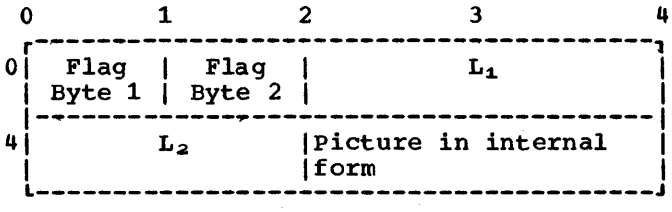
DED for FLOAT Data



DED for FIXED Data



DED for PICTURE STRING Data



Flag byte 1 = Hex 30

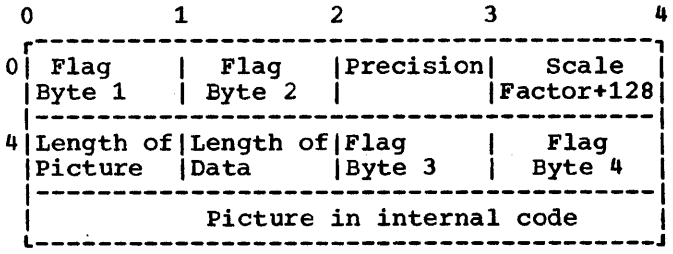
The internal code for string pictures is as follows:

Code	Picture(hex)
A	00
9	04
x	1C

L_1 = length of field with insertion characters

L_2 = length of field without insertion characters

DED for PICTURE DECIMAL Arithmetic Data



Flag byte 1 = Hex 14 or 1C

Flag Byte 3 (describes the mantissa subfield)

Bit 0 = reserved; must be set to '0'B

Bit 1 = '1'B drifting S in subfield
= '0'B no drifting S in subfield

Bit 2 = '1'B drifting + in subfield
= '0'B no drifting + in subfield

Bit 3 = '1'B drifting - in subfield
= '0'B no drifting - in subfield

Bit 4 = '1'B drifting $ in subfield
= '0'B no drifting $ in subfield

Bit 5 = '1'B total suppression in subfield
= '0'B no total suppression in subfield

Bit 6 = '1'B * in subfield
= '0'B no * in subfield

Bit 7 = reserved; must be set to '0'B

Flag Byte 4 (describes the exponent subfield)

Same format as Flag Byte 3.

Internal codes for pictures

Code	Picture	Code	Picture
00	9	48	- (t)
04	Y	4C	- (d)
08	Z	50	- (s)
0C	*	54	$ (t)
10	E	58	$ (d)
14	K	5C	$ (s)
18	T	60	/ (t)

1C	I	64	/ (d)
20	R	68	/ (s)
24	CR	6C	. (t)
28	DB	70	. (d)
2C	B	74	. (s)
30	S (t)	78	, (t)
34	s (d)	7C	, (d)
38	S (s)	80	, (s)
3C	+	84	V
40	+ (d)		
44	+ (s)		

(t) = terminal
(d) = drifting
(s) = static

Note: After E or K, the next byte contains the number of digits in the exponent.

Scale Factor

The scale factor of a picture DED is the number of digit positions after the 'V' (0 if there is no 'V') added to the number in the F specification, if any.

Rule for setting bit 5 in Flag Bytes 3 and 4

Bit 5 is set if no 9, Y, T, I, or R is present. This applies before any Z, S, etc. has been translated to a 9.

Rules for translating pictures into encoded pictures

1. Characters 9, Y, E, K, T, I, R, CR, DB, B, and V are translated directly.

2. Characters Z and * are translated directly if they do not follow a V. If either follows a V, it is translated into the code for character 9.

3. An S, +, -, or $ is translated to a static S, +, -, or $ if it is the only one of its kind in the subfield.

4. If more than one S appears in a subfield, the S's are translated into drifting S's.

Except when:

   a. It appears immediately before a Y, 9, V, T, I or R. In this case it is translated into the code for a terminal S.

   b. It appears anywhere after a V. In this case it is translated into the code for a 9.

   The same rule applies for the +, -, or $.

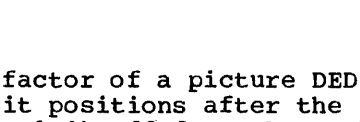
5. A "/", a ",", or a "." is treated as drifting, if:

   a. It is in a subfield containing either one or more Z or asterisk, or more than one +s, -s, or $.

and if:

   b. It is not immediately preceding a Y, 9, V, T, I, or R. In this case it is translated into terminal form.

DED for PROGRAM CONTROL Data

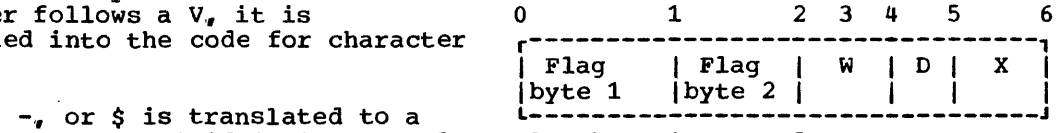


Flag byte 1 = Hex 80, 84, 88, 8C, 90, 94, 98, or 9C

FORMAT DEDS - FEDS

For meaning of flag bytes see above under Data Element Descriptors.

DED for F and E FORMAT Items (FED)



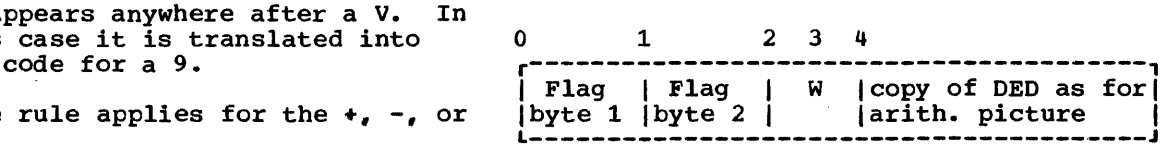
Flag byte 1 = Hex 50

W = total length of the format field

D = number of decimal places

X = precision + 128 for F-format number of significant figures for E-format

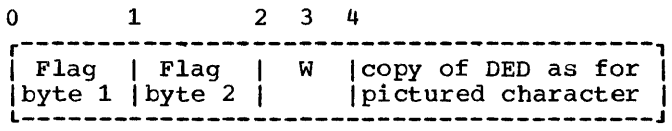
DED for PICTURE FORMAT Arithmetic Items (FED)



Flag byte 1 = Hex 54

W = total length of the format field

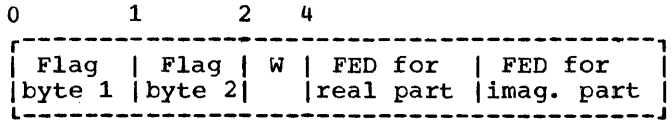
DED for PICTURE FORMAT character Items (FED)



Flag byte 1 = Hex 58

W = total length of the format field

DED for C FORMAT Items (FED)

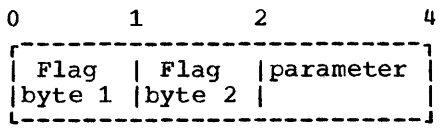


Flag byte 1 = Hex 5C

Note: The complex bit (bit 7) in flag byte 2 is set in both the real part and the imaginary part FED.

W = total length of the format field

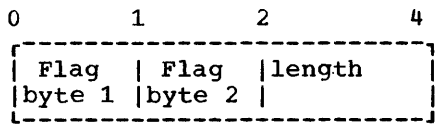
DED for CONTROL FORMAT Items (FED)



Flag byte 1 = Hex 60, 64, 68, 6C or 70

Parameter = length of item (X format)
column number (COL format)
number of lines to skip (SKIP format)
line number (LINE format)
is omitted for PAGE format

DED for STRING FORMAT Items (FED)



Flag byte 1 = Hex 58

The difference between A, B, and P (character) formats is given by bits 0 and 1 of flag byte 2. The length field may be omitted for A and B format items.

Declare Control Block (DCLCB)

Function

Addresses file via PRV, holds declared file attributes, filename, and address of ENVB.

When Generated

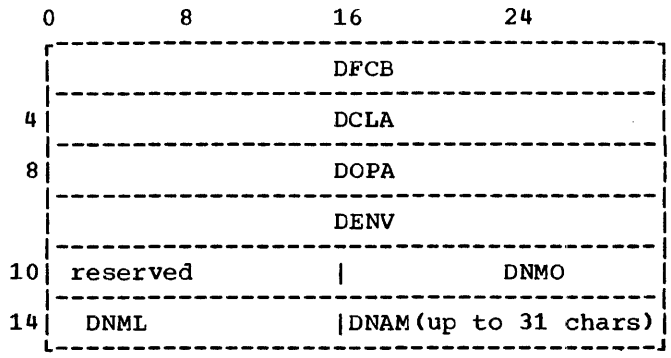
During compilation.

Where Held

In separate control section for external files, or static internal control section for internal files.

How Addressed

Address generated by linkage editor for external files addressed by an offset from register 3 for internal files.



DFCB A(FCB) or Pseudo-Register offset (in first 2 bytes)

DCLA Declare Attributes

Byte Number	Hexadecimal Value	Attributes
1	01	STREAM
	02	RECORD
	04	DISPLAY
	10	reserved
		(STRING)
2	01	SEQUENTIAL
	02	DIRECT
	04	TRANSIENT
	10	INPUT
	20	OUTPUT
	40	UPDATE
	80	BACKWARDS
3	01	BUFFERED
	02	UNBUFFERED
	04	KEYED
	08	EXCLUSIVE
	10	PRINT
4		reserved

DOPA Attributes which would conflict on OPEN. Format as for DCLA

DENV A(Environment Block) or zero

DNMO Offset in DCLCB of DNML

DNML Length of File name

DNAM File name (up to 31 characters)

Diagnostic File Block (DFB)

Function

Holds information used by the error message routines.

When Generated

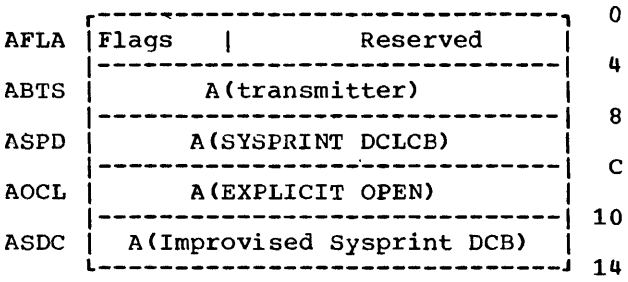
During program initialization.

Where Held

Program management area.

How Addressed

From X'40' in the TCA.



AFLA - Flags

AWTO Bit 0 = 1 Messages going to operator's console
ASNO Bit 1 = always 0
ASCO Bit 2 = 1 SYSPRINT cannot be opened or open with unsuitable attributes.
AFPF Bit 3 = 1 Force page

Dynamic Storage Area (DSA)

Function

Holds housekeeping information, automatic variables, and temporaries for each block.

When Generated

During execution. Allocated by prologue code every time a new block is entered.

Where Held

In the LIFO storage stack. Certain library routines have their DSAs in library workspace (LWS). See below

How Addressed

From register 13.

Flags

Bit 0 = 0 DSA in LWS
1 DSA

Bit 1 = 0 No ON Cells
1 ON cells

Bit 2 = 0 No Dynamic ONCBs
1 Dynamic ONCBs

Bit 3 Always set to zero.

Bits 4 and 5
= 00 Procedure DSA
01 Begin DSA
10 Library DSA
11 On-unit DSA

Bit 6 = 0 Not a dummy DSA
1 Dummy DSA

Bit 7 unused.

Bit 8 = always zero

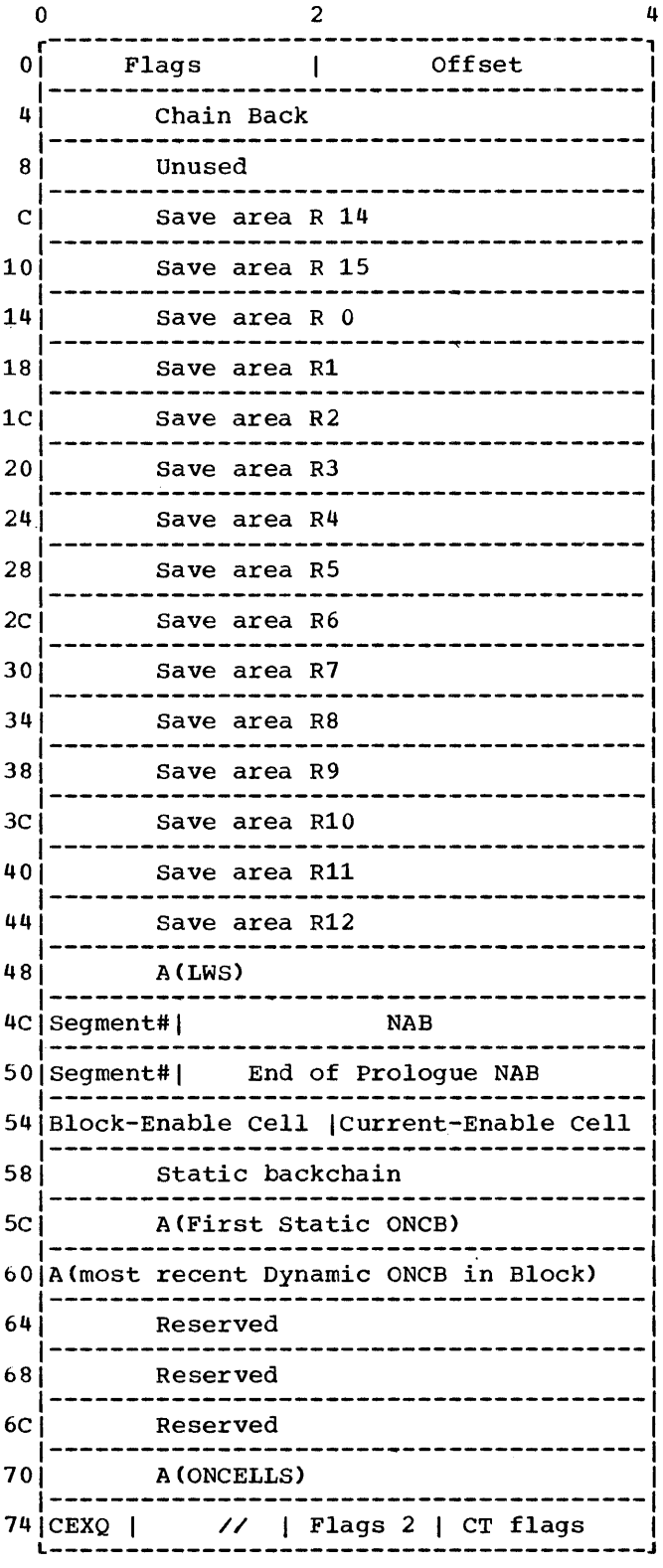
Bit 9 = 0 Do not restore NAB on GOTO
1 Restore NAB on GOTO

Bit 10 = 0 Do not restore Current-enable on GOTO
1 Restore current-enable on GOTO

Bit 11 = 0 Callee cannot use this DSA
1 Callee can use this DSA

Bit 12 = 0 Not an EXIT DSA
1 EXIT DSA

Bit 13 = 0 No statement # table
1 Statement # table available



Flags continued

Bit 14 = 1 Sysprint ENQ'd

Bit 15 = 1 Flags 2 valid

Offset

If the DSA is in LWS, offset is the offset of the ONCA. Otherwise, this field is not used.

CEXQ

Save area for flag byte 1 of the TCA. Used if DSA is an exit DSA.

Flags 2

Bit 0 = 1 Last PL/I DSA
Bit 1 = 1 Ignore DSA for SNAP
Bit 2 = 1 ILC DSA after interrupt
Bit 3 = 1 Invocation Count in this DSA
Bit 4 Reserved
Bit 5 = 1 There are TSO line numbers

CT (Control Task) Flags

Bit 0 = 1 Block has active subtasks

Note: This flag byte is the only one in the DSA used by the control task without synchronising with the subtask. The subtask must never change it. This prevents interference between CPU's on a multiprocessing machine.

Dump Block (DUB)

Function

To hold information about the dump file.

When Generated

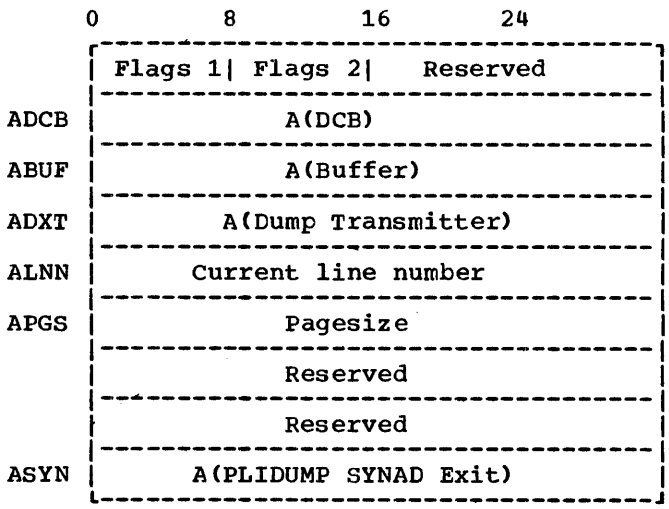
During program initialization.

Where Held

In the program management area.

How Addressed

From offset X'120' in the TIA.



Flags 1

ANDE Bit 0 = 1 Dump file cannot be opened

Flags 2

ANSS Bit 0 = 1 No subtasks' subpools

Entry Data Control Block

Function

Holds the addresses of the data item and its DSA.

When Generated

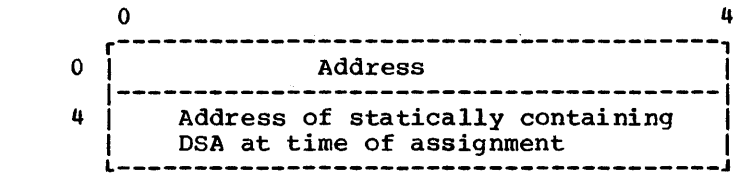
When the variable is allocated.

Where Held

Depends on the storage class of the data item.

How Addressed

Depends on the storage class of the data



Word 1: bit 0 = 0 Address of entry
= 1 Address of location containing 8-char. EBCDIC name of entry point

Word 2: bit 0 always = 0

Environment Block (ENVB)

Function

Holds environment options for a file so that the file may be correctly opened during execution.

When Generated

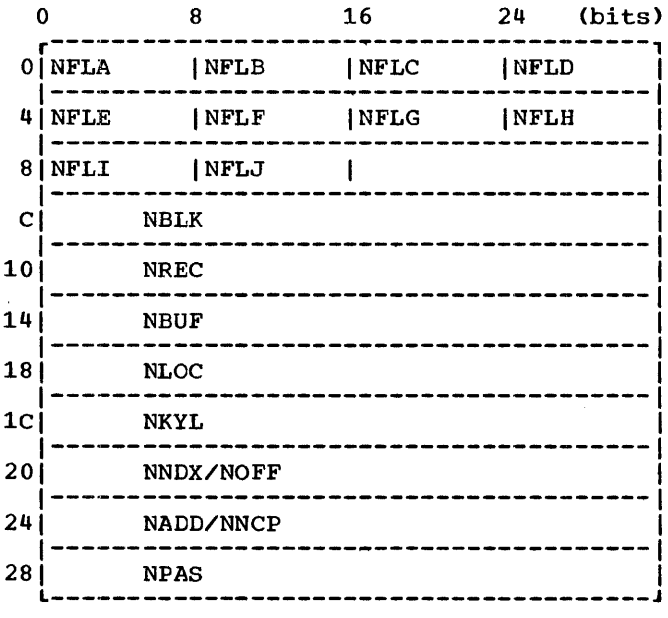
During compilation

Where Held

In a control section with the DCLCB for external files. In static internal storage for internal files.

How Addressed

From offset X'C' in the DCLCB



NFLA
- 0 consecutive
- 1 indexed
- 2 regional (1)
- 3 regional (2)
- 4 regional (3)
- 5 tp(m)
- 6 tp(r)
- 7 Other organization see NFLH

NFLB
Bits 0 & 1
- 10 Fixed
- 01 Variable
- 11 Undefined

Bit
- 2 D or TRKOFL
- 3 Blocked
- 4 Spanned
- 5 CTLASA
- 6 CTL360
- 7 Unused

NFLC
- 0 LEAVE
- 1 REREAD
- 2 GENKEY
- 3 COBOL
- 4 NOWRITE
- 5 INDEXAREA
- 6 TOTAL
- 7 INDEXAREA with no argument

NFLD
- 0 BUFFERS
- 1 NCP
- 2 Unused
- 3 KEYLENGTH
- 4 KEYLOC
- 5 VERIFY
- 6 NOLABEL
- 7 ADDBUF

NFLE
- 0 Unused
- 1 Unused
- 2 Unused
- 3 Unused
- 4 SCALARVARYING
- 5 ANSCII
- 6 BUFOFF
- 7 BUFOFF(L)

NFLF reserved

NFLG
- 0 F-format
- 1 V-format
- 2 U-format
- 3 Spanned
- 4 Blocked
- 5 Unused
- 6 Unused
- 7 Unused

NFLH
- 0 VSAM
- 1-7 Reserved

NFLI, NFLJ reserved

NBLK A(blocksize)

NREC A(record length)

NBUF A(number of buffers)

NLOC A(KEYLOC value)

NKYL A(KEYLENGTH)

NNDX A(INDEXAREA size)
NOFF A(BUFOFF value)

NADD A(size of ADDBUF)

NNCP A(NCP value)

NPAS A(password string locator)

Event Table (EVTAB)

Function

Used by WAIT module as workspace and to provide status information on associated event.

When Generated

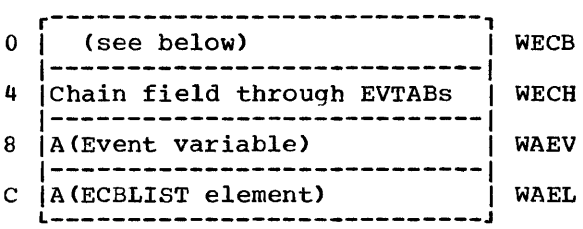
During execution.

Where Held

In LIFO storage.

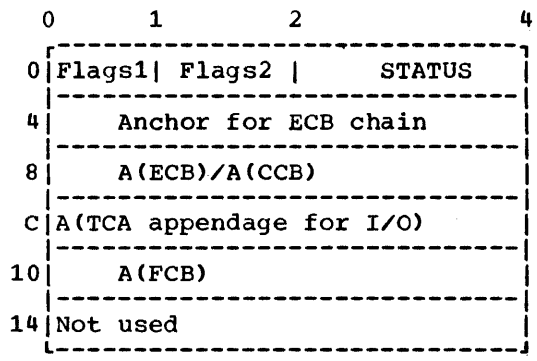
How Addressed

From an offset from register 13.



WECB Bit 0 set when event is complete
Bits 1-7 Not used in this implementation

Event Variable Control Block



Function

To hold information about the operation with which the EVENT has been associated.

When Generated

Depends on the storage class of the event variable.

Where Held

Depends on the storage class of event variable.

How Addressed

As other variables depending on storage class.

Flags 1

Bit 0 =0 Incomplete
1 Complete

Bit 1 =0 Inactive
1 Active

Bit 2 =0 Not an I/O EVENT
1 I/O EVENT

Bit 3 =0 Not a DISPLAY EVENT
1 DISPLAY EVENT

Bit 4 =0 EV has not caused on-unit entry
1 EV has caused entry to an on-unit

Bit 7 =always zero

Flags_2

Bit 0 =0 No chain of ECBs
1 Chain of ECBs exists

Bit 1 =0 Not a dummy EVENT
1 Dummy EVENT

Exclusive Block IOCB (XBI)

Function

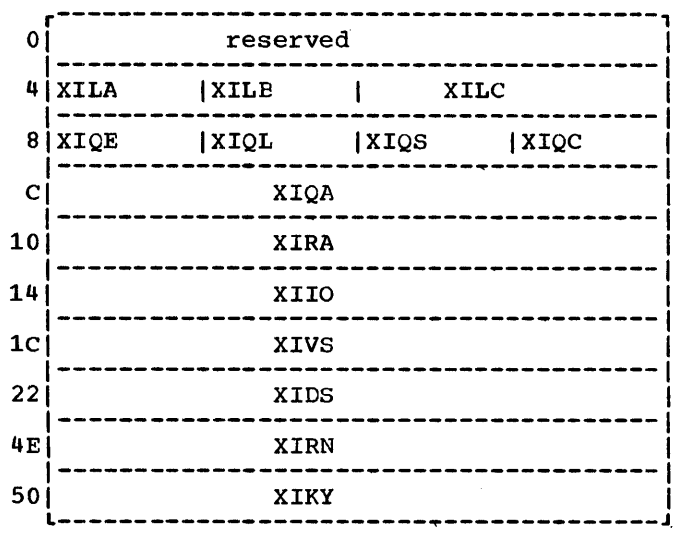
Locks individual records on exclusive files.

When Generated

By transmitter when required

How Addressed

From offset X'24' in IOCB and offset X'14' in the TIA.



First two words unused

XIQE Start of ENQ LIST for system X'FF'

XIQL Length of RNAME

XIQS System flags must be X'41'

XIQC Return code from system

XIQA Address of QNAME (XFIO)

XIRA Address of RNAME (XFVS)

XIIO QNAME ('SYSIBMIO')

XIVS RNAME of volume serial no. (part 1)

XIDS RNAME of DSNAME (part 2)

XIRN region no. in binary right adjusted

XIKY RNAME of key (part 3)

Length of XIKY is keylength of data set restricted such that volume serial no.||dsname||key < 255 ISAM
251 regional

Exclusive Block File (XBF)

Function

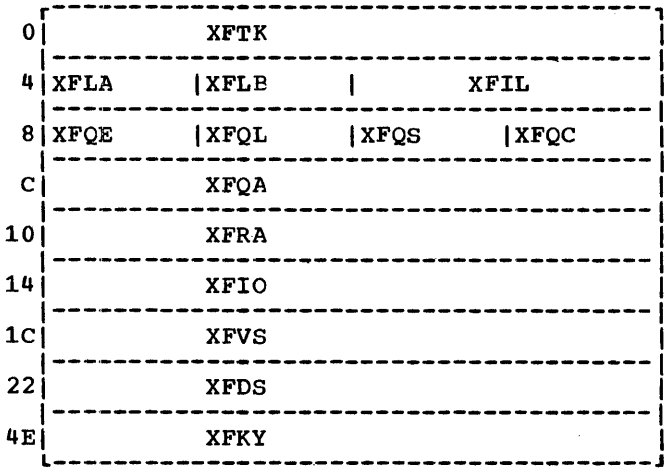
Identifies data set when locking for exclusive I/O.

When Generated

By the open routine

How Addressed

From offset X'74' in FCB.



First two words reserved

XFTK A(TCA)

XFLA First flag byte

Bit 0 = 1 Locked
Bit 1 = 1 No DEQ required

XFLB second flag byte (reserved)

XFIL Length of exclusive block attached to IOCB

XFQE Start of ENQ LIST for system X'FF'

XFQL Length of RNAME

XFQS System flags must be X'41'

XFQC Return code from system

XFQA Address of QNAME (XFIO)

XFRA Address of RNAME (XFVS)

XFIO QNAME ('SYSIBMIO')

XFVS RNAME of volume serial no. (part 1)

XFDS RNAME of DSNAME (part 2)

XFKY RNAME of key (part 3)

Length of XFKY is keylength of data set restricted such that volume serial no.||dsname||key < 255 ISAM
251 regional

File Control Block (FCB)

Function

Used to access all file information. Contains addresses of the ENVB,DTF, filename, etc.

When Generated

By the open routines during execution.

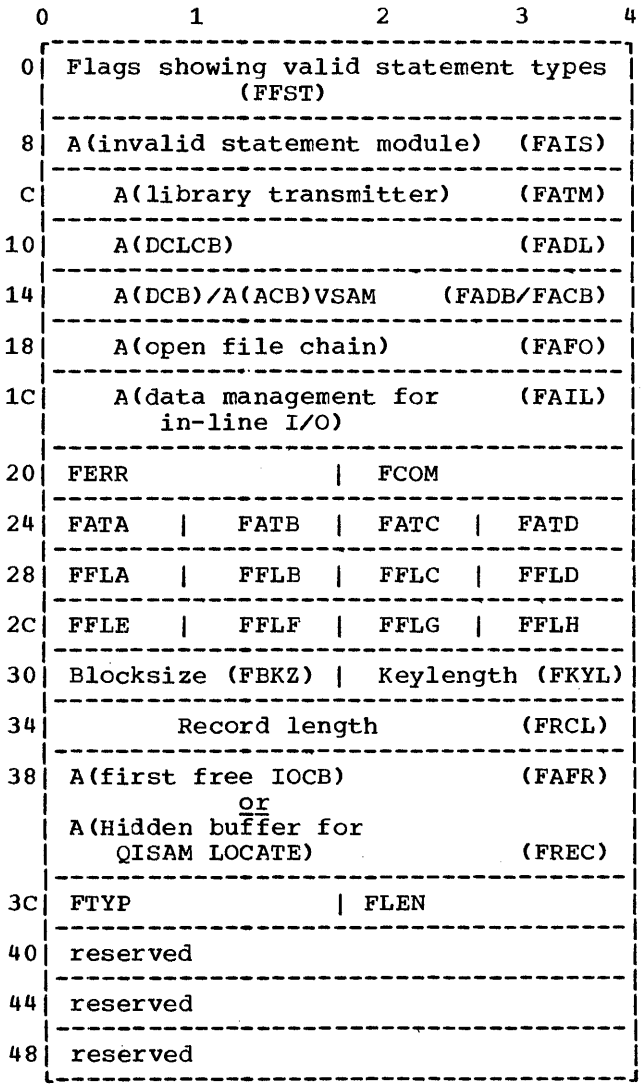
Where Held

In subpool 1.

How Addressed

From two byte PRV offset which is held at offset X'0' in DCLCB. The PRV address is held at offset X'4' in the TCA.

Common Section



FFST Flags indicating types of statement (8 bytes)

Bit number	Statement + options
0	READ SET
1	READ SET KEYTO
2	READ SET KEY
3	READ INTO
4	READ INTO KEYTO
5	READ INTO KEY
6	READ INTO KEY NOLOCK
7	READ IGNORE
8	READ INTO EVENT
9	READ INTO KEYTO EVENT
10	READ INTO KEY EVENT
11	READ INTO KEY NOLOCK EVENT
12	READ IGNORE EVENT
13	WRITE FROM
14	WRITE FROM KEYFROM
15	WRITE FROM EVENT
16	WRITE FROM KEYFROM EVENT
17	REWRITE
18	REWRITE FROM
19	REWRITE FROM KEY
20	REWRITE FROM EVENT
21	REWRITE FROM KEY EVENT
22	LOCATE SET
23	LOCATE SET KEYFROM
24	DELETE
25	DELETE KEY
26	DELETE EVENT
27	DELETE KEY EVENT
28	UNLOCK KEY
29-63	Reserved

FERR Error codes

X'02'	INPUT TRANSMIT (DATA SET)
X'03'	OUTPUT TRANSMIT (DATA SET)
X'1A'	OMR READ ERROR
X'1C'	INPUT TRANSMIT (INDEX SET)
X'1D'	OUTPUT TRANSMIT (INDEX SET)
X'1E'	INPUT TRANSMIT (SEQUENCE SET)
X'1F'	OUTPUT TRANSMIT (SEQUENCE SET)
X'01'	END OF FILE
X'04'	ZERO LENGTH RECORD VARIABL
X'05'	SHORT RECORD VARIABLE
X'06'	LONG RECORD VARIABLE
X'07'	KEY CONVERSION IN CHAR STRING
X'08'	KEY DUPLICATION
X'09'	KEY SEQUENCE
X'0A'	KEY SPECIFICATION (NULL KEY)
X'0B'	KEY NOT FOUND
X'0cC'	NO SPACE FOR KEYED RECORD
X'0D'	NO IOCB AVAILABLE
X'0E'	ACTIVE EVENT
X'0F'	NO PRIOR READ BEFORE REWRITE
X'10'	NO COMPLETED READ BEFORE REWRITE
X'11'	PERMANENT OUTPUT ERROR
X'12'	ZERO LENGTH RECORD READ
X'13'	REC. REFERENCE OUTSIDE DATA SET
X'14'	UNIDENTIFIED IO ERROR
X'15'	INCOMPLETE READ FOR UPDATE
X'16'	TP TERM ADDR SPECIFICATION
X'17'	DIFF FCB SAME RECORD REQUEST
X'18'	KEY CONVERSION (NEG BIN NC)
X'19'	KEY SPECIFICATION (X'FF' ETC)

X'1B'	I/O SEQUENCE ERROR	
X'20'	SYNAD ERROR ENCOUNTERED	
X'21'	RECORD LENGTH < KEYLEN + RKP	
X'22'	RECORD ALREADY HELD	
X'23'	RECORD ON NON-MOUNTED VOLUME	
X'24'	DATA SET CANNOT BE EXTENDED	
X'25'	NO VIRTUAL STORAGE FOR VSAM	
X'26'	NO KEYRANGE FOR INSERTION	
X'27'	NO POSITIONING FOR SEQL READ	
X'28'	ATTEMPT TO REPOSITION FAILED	

FCOM Reserved for future releases for compatibility flags.

FTYP 6th and 7th characters of library transmitter name

FLEN Length of FCB

FATA-FATD Flags showing attributes allowable with file types, and other file usage information.

	Bit	Attribute
FATA	0	Open SYSPRINT for error message
	1	SYSPRINT
	2	unused
	3	String operation
	4	unused
	5	unused
	6	RECORD
	7	STREAM
FATB	0	BACKWARDS
	1	UPDATE
	2	OUTPUT
	3	INPUT
	4	unused
	5	TRANSIENT
	6	DIRECT
	7	SEQUENTIAL
FATC	0	unused
	1	unused
	2	unused
	3	PRINT
	4	EXCLUSIVE
	5	KEYED
	6	UNBUFFERED
	7	BUFFERED
FATD	all	unused

	Bit	
FFLA	0	F-format
	1	V-format
	2	U-format
	3	Blocked
	4	Spanned
	5	unused
	6	unused
	7	Key in record variable KEYLOC
FFLB	0	CONSECUTIVE
	1	INDEXED
	2	REGIONAL(1)
	3	REGIONAL(2)
	4	REGIONAL(3)
	5	TP(M)
	6	TP(R)
	7	Other organization

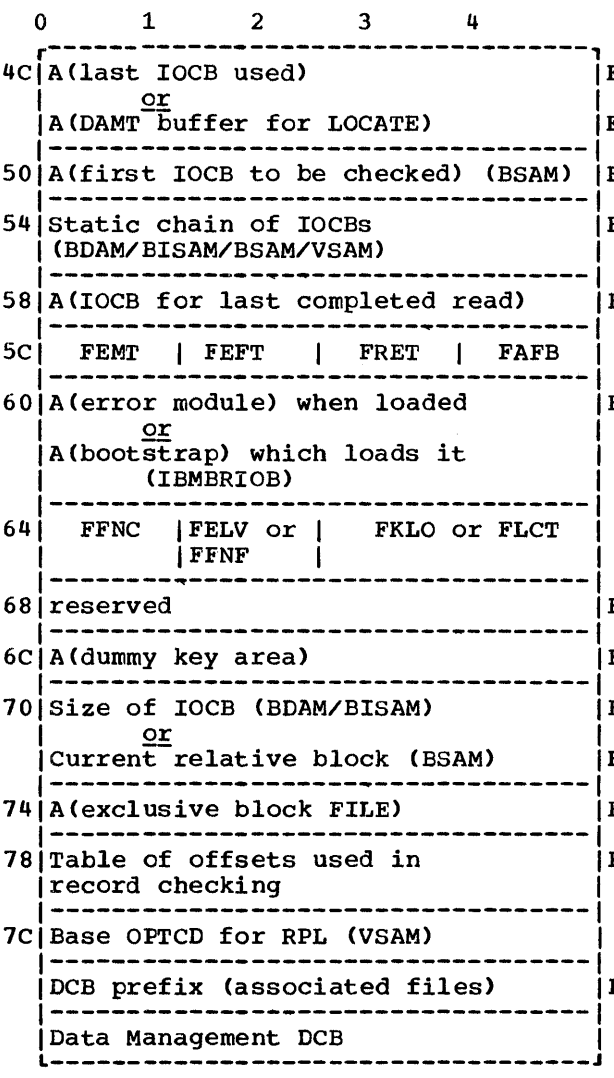
	Hex	Contents
FFLC	0	QSAM
	4	BSAM
	8	BSAM(Load)
	0C	TCAM
	10	QISAM
	14	BISAM
	18	BDAM
	1C	VSAM

	Bit	
FFLD	0	Paper tape
	1	Printer
	2	Unit record device
	3	The Foreground Terminal
	4	ENDFILE module loaded
	5	Possible hidden buffer
	6	Error module loaded
	7	Genkey
FFLE	0	I/O error
	1	permanent input error
	2	permanent output error
	3	end of file
	4	hidden buffer in use
	5	move required
	6	non-SCALARVARYING
	7	reserved
FFLF	0	previous READ
	1	previous READ SET
	2	previous LOCATE
	3	previous REWRITE
	4	previous OPEN
	5	close in progress
	6	implicit close
	7	previous OPEN(resume load)
FFLG	0	ENDPAGE
	1	end of extents
	2	COPY option active
	3	reserved
	4	checkout transmitter
	5	checkout compiler step end flag
	6	newly opened print file
	7	file not to be closed
FFLH	0	In-line I/O
	1	In-line locate
	2	hyphen at end of line
	3	retry get after concat
	4	current line unfinished
	5	initial call from IBMBSPL/ blanks at end of record
	6	new buffer wanted
	7	GET prompt issued - input

The common section is followed by either the RECORD or STREAM sections.

Record I/O Section

Note: Offsets are from start of the FCB.



FRET Data management return code (regional output)

FEMT 6th char of error module name

FEFT 7th char of endfile module name

FAFB Work byte for associated files

FFNC Function byte

Bit 0 READ file
1 PUNCH file
2 PRINT file
3 OMR (no other lists on)
4 R in FUNC option
5 P in FUNC option
6 W in FUNC option
7 Associated file

FFLV VSAM flags

Bit 0 KSDS
1 ESDS
2-7 reserved

FCNF conflict byte

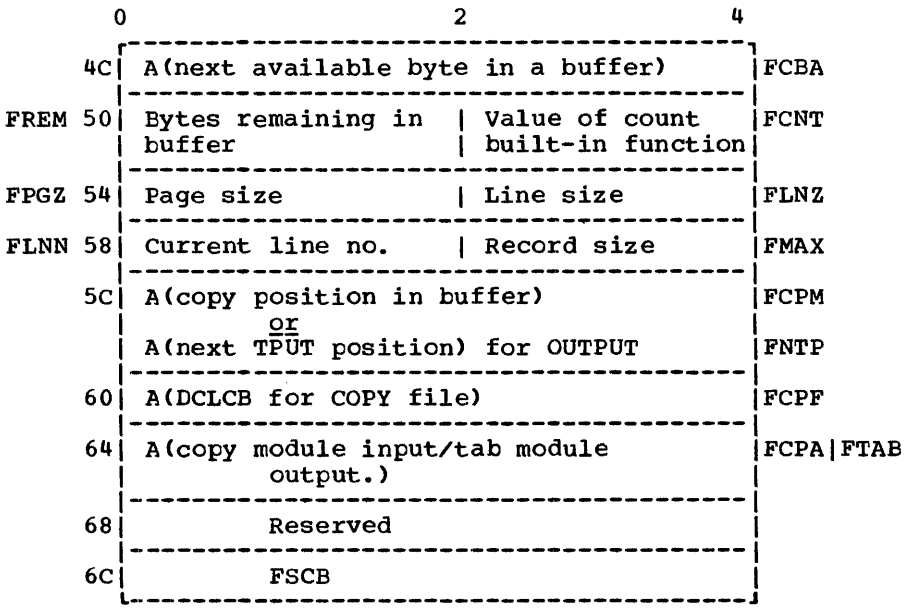
Bit 0 prior READ invalid
1 prior PUNCH invalid
2 prior PRINT invalid
3 prior PRINT last line invalid
4-7 reserved

FKLO KEYLOC-1

FLCT Decrementing line count

Stream I/O Section

Offsets are from the start of the FCB.



Flow Statement Table

Function

Used to implement the compiler FLOW option. Holds the last 'n' statement number pairs and the last 'm' procedure names executed. ('n' and 'm' are programmer defined.)

When Generated

Storage is allocated during initialization if the FLOW option has been specified. The table is continually updated as the program is executed.

Where Held

In initial storage area.

How Addressed

From offset X'4C' in the TCA.

AFLG-Flags

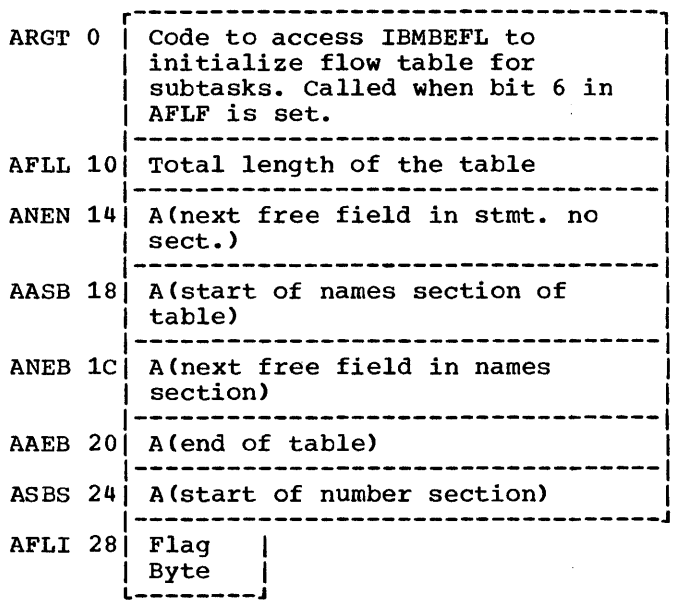
Reserved

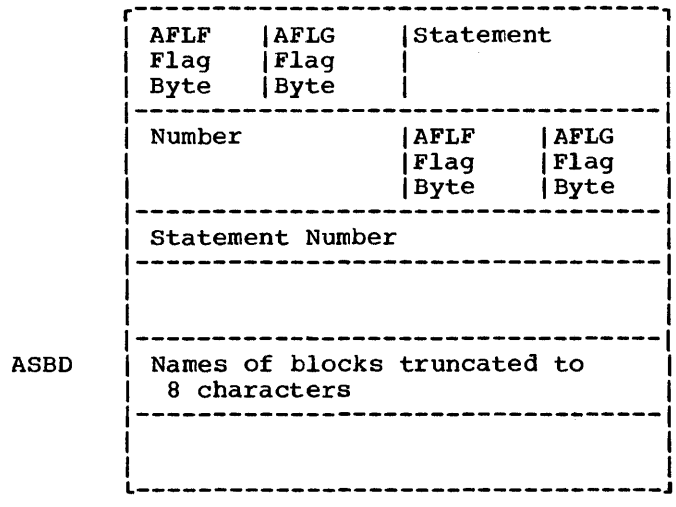
AFL1-Flag

Bit 0 = 1 No statement numbers requested in flow trace (e.g. FLOW(0,20)).
1 = 1 Last entry was in.
2 = 1 Used by Checkout Compiler only.
3 = 1 Interrupt not recorded.
4 = 1 GOTO-out-of-block has occurred.

AFLF - Flags

ATBI Bit 0 Branch-in entry
ABCD Bit 1 BCD form for this entry
ATXT Bit 2 BCD in text reference form
ADUM Bit 3 Dummy entry after on-unit exit
ACHK Bit 4 unused
AGTO Bit 5 Unused
ATKC Bit 6 The next in entry is in new task





Interlanguage Root Control Block (IBMBILC1)

Function

Connects ZCTL and interlanguage VDA to interlanguage routines, and records state of activation of language interfaces.

When Generated

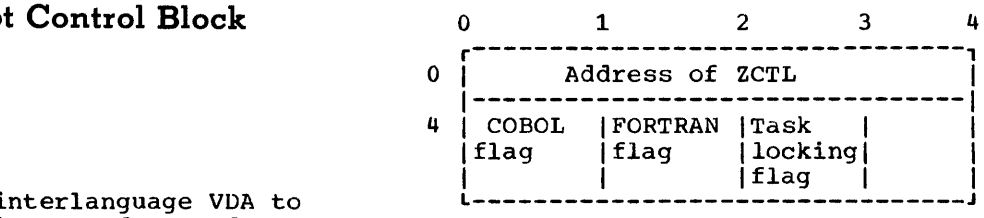
During compilation.

Where Held

In static internal storage, as a control section.

How Addressed

Address generated by linkage editor.



Flags

COBOL flag indicates COBOL is active in program

FORTRAN flag indicates a procedure which called FORTRAN is active

Task locking flag indicates that a task is accessing IBMBILC1

Interlanguage VDA

Function

To hold information required for interlanguage calls. Used for information that alters from invocation to invocation.

When Generated

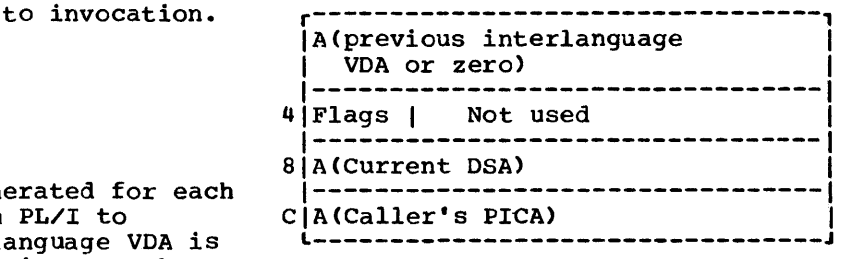
One interlanguage VDA is generated for each interlanguage call made from PL/I to FORTRAN or COBOL. An interlanguage VDA is also acquired if the PL/I environment has not yet been set up when PL/I is called from COBOL or FORTRAN.

Where Held

In the LIFO storage stack.

How Addressed

From offset X'0' in ZCTL.



Flags

Bit 0 = 1 If there is a previous call to COBOL
1 = 1 If there is a previous call to FORTRAN
2 = 1 If main procedure is not PL/I

Interrupt Control Block (ICB)

Function

Acts as a parameter list to IBMBERR.

When Generated

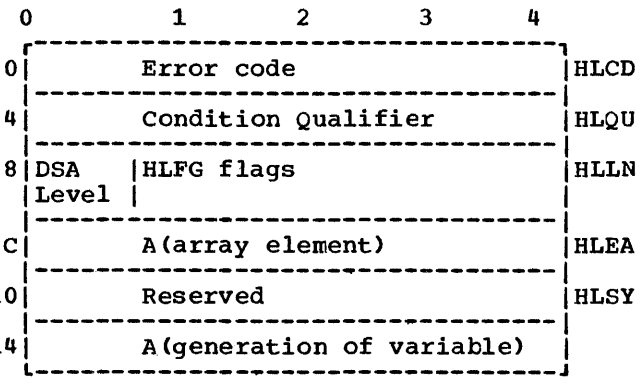
After an error has been detected.

Where Held

As a VDA in the LIFO stack.

How Addressed

Passed as parameter list to IBMBERR addressed by register 1.



Condition qualifier = A(DCLCB) for I/O condition
= A(CSECT) for CONDITION condition
= A(SYMTAB) for CHECK condition
= A(SYMTAB LIST)

HLFG flags

Bit 0 = 0 Reserved

Bit 1 = 0 Element address not in list
1 Element address in list

Bit 2 = 0 CHECK is enabled
1 CHECK enablement unknown

Bit 3 = 0 Qualifier is not address of SYMTAB list
1 Qualifier is address of SYMTAB list

Bit 4 = 0 Not word 6 information
1 Use word 6 to address the generation of variable being checked

Input/Output Control Block (IOCB)

Function

Used as a data management parameter list during certain record I/O statements and to hold information about statement type during the time between a record I/O statement and the associated WAIT statement.

When Generated

Either by the PL/I transmitter module (BISAM or BDAM) or by the OPEN module.

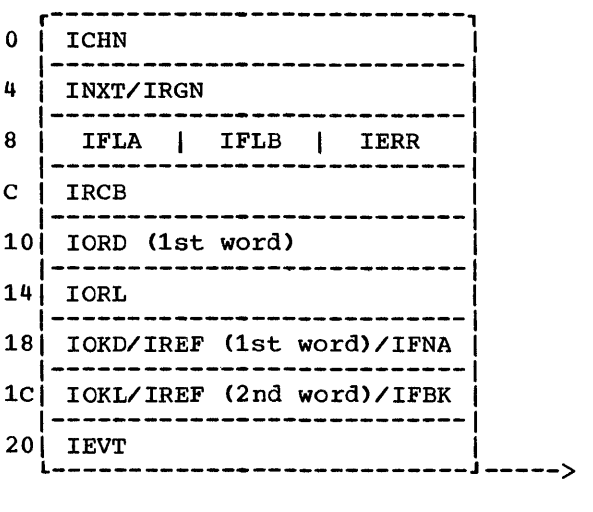
Where Held

In non-LIFO storage for VSAM, in subpool 0 for BSAM (obtained by GETPOOL), BISAM or BDAM (obtained in non-tasking, in subpool 0 for tasking.

How Addressed

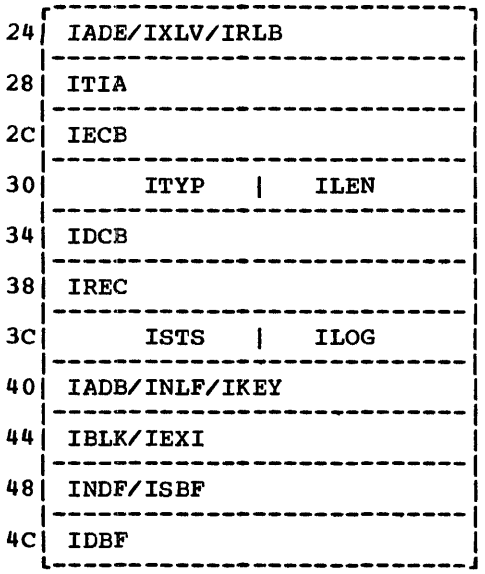
By fields in the FCB. IOCBs are chained together and the actual field used to address them depends on the type of statement being executed.

Common Section



ICHN Static forward chain
INXT Chain of free or unchecked 10lbs.
IRGN Region no. left adjusted (BDAM)
IFLA Flag byte - bits set to '1' indicate:
Bit 0 = record locked
1 = record to move
2 = varying string with non-scalarvarying
3 = IOCB in use
4 = general error flag
5 = dummy records being output
6 = dummy buffer acquired
7 = IOCB checked
IFLB Code byte containing offset within 'look-up' table used for record checking
IERR Error codes (as in FERR of FCB) first byte is for TRANSMIT, second byte for ENDFILE, RECORD, KEY & ERROR conditions.
IRCB Request Control Block
IORD 1st word of record description - Record address
IORL 2nd word of record descriptor - flags and record
IOKD 1st word of key description - key address length by region number
IOKL 2nd word of KD - flags and key length
IREF Relative block in record numbers (2 words) (BDAM)
IFNA Next address feedback (Regional 3, spanned)
IFBK BDAM feedback (BDAM spanned)
IEVT A (EVENT variable)

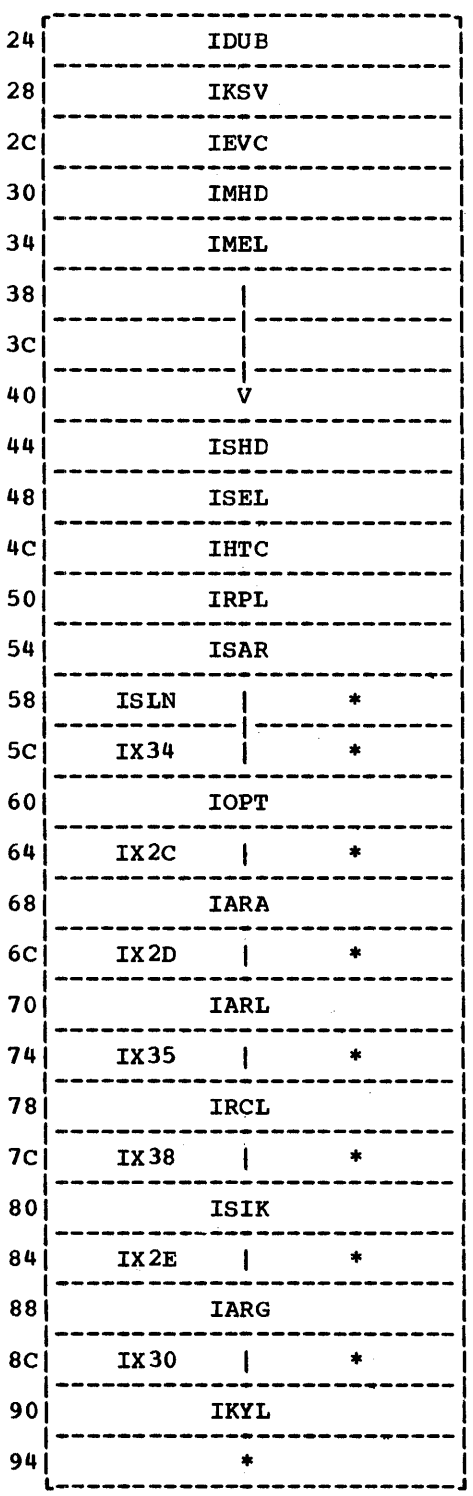
Second section for non VSAM files



IADE A (ECB) for REGIONAL SEQUENTIAL ONLY
IXLV A (Exclusive block) for direct only

IRLB Binary region no. (Regional(1)Update)
ITIA A (Implementation Appendage)
IECB Data management Event Control Block (BDAM exception codes in 1st 2 bytes)
ITYP Type of I/O operation (set by Data management)
ILEN Record length
IDCB A(DCB)
IREC A (buffer) if one exists on A (record variable)
ISTS A (status indicators) (BSAM & BDAM)
ILOG A (logical record) (BISAM)
IADB A(dummy buffer) (BSAM)
INLF A (next record feedback) --> IREF (BSAM)
IKEY A (KEY) (BDAM & BISAM)
IBLK A (relative block or record) i.e. A (IREF) (BDAM)
IEXI BISAM exception codes
INDF A (next record feedback) --> IREF (BDAM)
ISBF Start of appended buffer (BSAM)
IDBF Start of appended buffer (BDAM & BISAM)

Second section for VSAM files



*Reserved fields

VSAM Section (starting at offset X'24')

IDUB A (dummy buffer)
IKSV A (key save user)
IEVC Data Management Event Control Block

MODCB plist (5 words starting at offset X'30')
IMHD A (header entry) --> IHTC
IMEL Element entry addresses (maximum of 4)

SHOWCB plist (2 words starting at offset X'44')
ISHD A (header entry) --> IHTC
ISEL A (element entry)

Header control entry (4 words starting at offset X'4C')
IHTC header type code for MODCB|SHOWCB of RPL
IRPL A (Request Parameter List)
ISAR A (receiving area for SHOWCB)
ISLN L (receiving area for SHOWCB)

Element control entries start at offset X'5C' and continue to end of IOCB. Each entry occupies 2 words, with keyword type code set in 1st half-word as follows: IXab = X'00ab' 2nd word of each entry is used as either a setting field for MODCB or a receiving field for SHOWCB. The IOCB field names are lifted with their corresponding RPL (Request Parameter List) parameters.

IOPT OPTCD
IARA AREA
IARL AREALEN
IRCL RECLEN
ISIK FDBK
IARG ARG
IKYL KEYLEN

Key Descriptor (KD)

Function

Contains address and length of key for passing to library record I/O routines.

When Generated

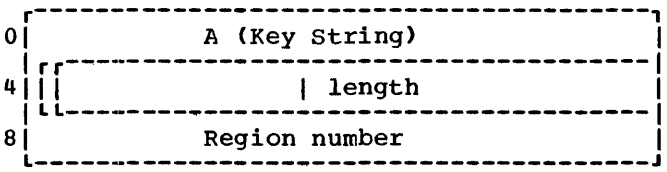
As far as possible during compilation. If necessary, completed during execution.

Where Held

Normally in static internal control section. In static external control section if key is EXTERNAL. Will be copied into, or generated in, temporary storage if procedure is reentrant or recursive.

How Addressed

From an offset from register 3 known to compiler code for internal keys.



Word 1

KEYFROM: Address of source key (excluding the length bytes if VARYING)

KEYTO: Address of where to put key (excluding length bytes if VARYING)

Word 2

Bit 0	'1'B if KEYTO string is VARYING. (If this bit is set, the I/O transmitters will set the current length field).
Bit 1	'1'B if word 3 contains a region number.
Bits 2-15	Unused (zero)
Bits 16-31	Length of key string (excluding length bytes for VARYING); current length for KEY or KEYFROM, maximum length for KEYTO.

Word 3

Region number in fixed binary, right justified.

Label Data Control Block

Function

Holds the address of the data item and, if a label variable, the address of the associated DSA.

When Generated

Label constants: during compilation
Label variables: when the variable is allocated depending on storage class.
Label temporaries: When required for GOTO to label constant.

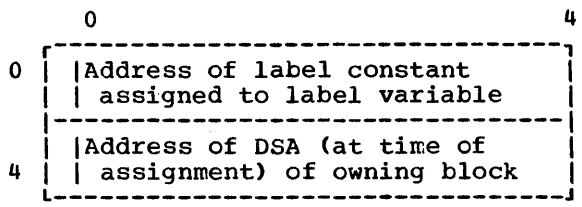
Where Held

Depends on the storage class of the data item

How Addressed

As a variable.

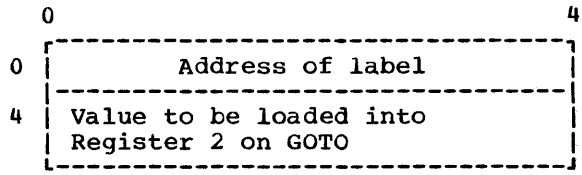
Label Variable and Label temporary



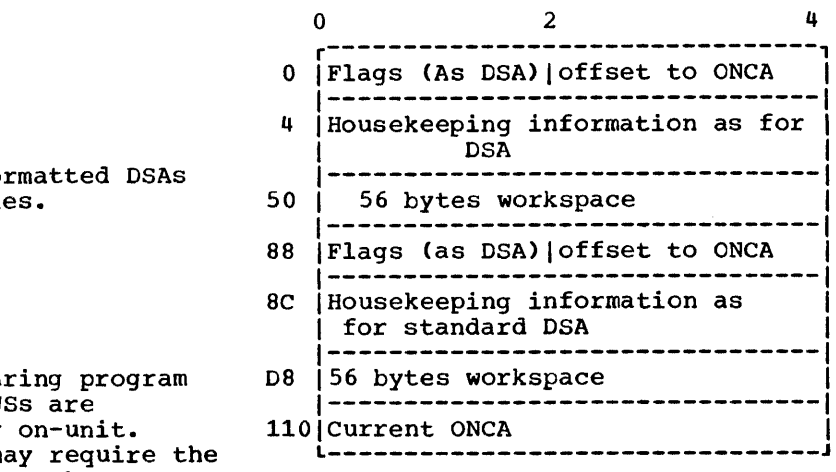
Word 1: bit 0 = 0 Address of label constant
= 1 Text reference

Word 2: bit 0 always = 0

Label Constant



Library Workspace (LWS)



Function

Space reserved for two pre-formatted DSAs used by certain library modules.

When Generated

The first LWS is generated during program initialization. Subsequent LWSs are allocated before entry to any on-unit. This is because the on-unit may require the use of library modules using LWS but must not alter the environment of the interrupt.

Where Held

First allocation in the program management area. Subsequent allocations in the LIFO storage stack. ONCAs are generated with LWS.

How Addressed

From offset X'48' in each DSA.

On Communications Area (ONCA)

Function

An area in which built-in function values or their addresses are placed, after the occurrence of a PL/I interrupt.

When Generated

The first ONCA is generated during program initialization. Subsequent ONCAs are generated with each allocation of LWS.

Where Held

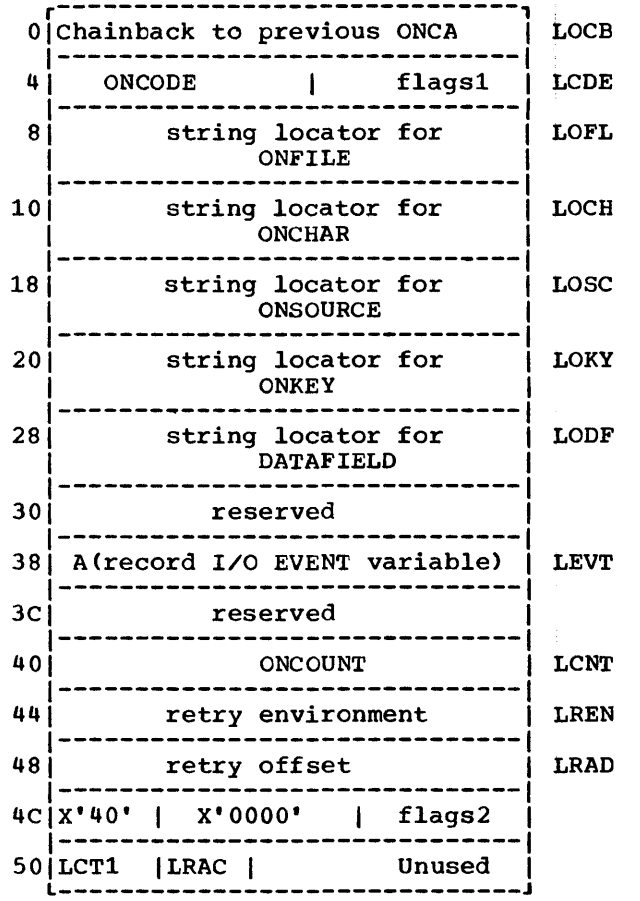
Contiguous with LWS in the program management area and in the LIFO stack.

How Addressed

By an offset from the current generation of library workspace. The offset is held as a halfword at offset X'2' in LWS.

Dummy ONCA

The dummy ONCA has the same format as other ONCAs and holds default values for those condition built-in functions that have default values.



Flags1

Bit 0 = 0 ONFILE invalid
= 1 ONFILE valid

Bit 1 = 0 ONCHAR/ONSOURCE invalid
= 1 ONCHAR/ONSOURCE valid

Bit 2 = 0 ONIDENT invalid
= 1 ONIDENT valid

Bit 3 = 0 ONKEY invalid
= 1 ONKEY valid

Bit 4 = 0 DATAFIELD invalid
= 1 DATAFIELD valid

Bit 5 = 0 No associated EVENT variable
= 1 Associated EVENT variable

Bit 6 = 0 ONATTN invalid
= 1 ONATTN valid

Bit 7 = 0 ONCOUNT invalid
= 1 ONCOUNT valid

Bits 8-15 unused

Flags_2

Bit 0 = 0 ONSOURCE/ONCHAR not used in on-unit
= 1 ONSOURCE/ONCHAR used in on-unit

Bit 1 = 0 ONSOURCE not set in ONCA
= 1 ONSOURCE set in ONCA

Bits 2-7 unused

LCT1

Copy of TCA flag byte 1 (TFB1)

LRAC

Retry address code

Retry_offset

The offset from the base of the library module involved to the address at which a conversion will be reattempted if ONSOURCE or ONCHAR has been used.

On Control Block (ONCB)

Function

Contains pointer to associated on unit, or indicates action to be taken when interrupt occurs.

How Addressed

From offset X'60' in the TCA.

When Generated

Static ONCBs are generated during compilation, one for each ON statement. Dynamic ONCBs are generated by the prologue code of the procedure or block in which the ON statement occurs, or are allocated in a VDA when the ON statement is executed.

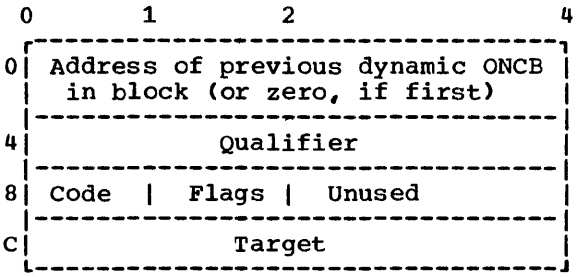
Where Held

Static ONCBs are generated in the static internal control section. Dynamic ONCBs are stored in the DSA of the block in which the associated on-unit occurs.

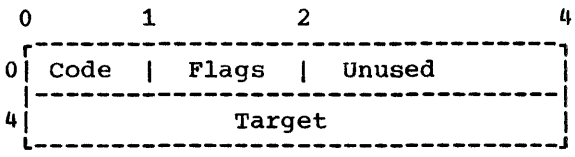
Static and Dynamic ONCBs

Static ONCBs are generated for unqualified conditions. Dynamic ONCBs are generated for qualified conditions (ENDPAGE, ENDFILE, etc.)

Dynamic ONCB



Static ONCB



Qualifier

A(FCB) for I/O conditions A(SYMTAB) for CHECK A(CSECT) for CONDITION condition.

Code

PL/I code for condition

Flags

Bit 0 = SYSTEM not specified
1 SYSTEM specified

Bit 1 = Not a null on-unit
1 Null on-unit

Bit 2 = Not a GOTO only on-unit
1 GOTO only on-unit

Bit 3 = Condition not established
1 Condition established

Bit 4 Unused

Bit 5 = Condition not enabled at block entry
1 Enabled at block entry.

Bit 6 = Condition disabled
1 Condition enabled

Bit 7 = SNAP not specified
1 SNAP specified

Target

Address of on-unit, or Offset in DSA of word containing A(label variable)

Open Control Block (OCB)

Function

Used to indicate that a file attribute (either input or output) was declared in the associated OPEN statement.

When Generated

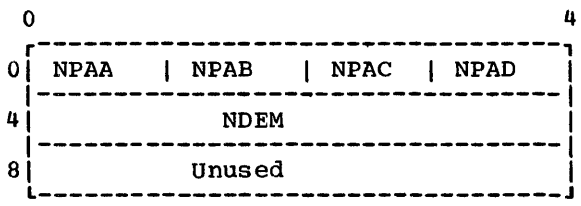
During compilation.

Where Held

Static internal control section.

How Addressed

From an offset from register 3 known to compiled code.



NPA - Open attributes

This word indicates the explicit and implied attributes on the OPEN statement.

Byte No.	Hex. Value	Attributes
1	01	STREAM
	02	RECORD
	04	DISPLAY
	10	reserved (STRING)
	80	Debug open of SYSPRINT
2	01	SEQUENTIAL
	02	DIRECT
	08	TRANSIENT
	10	INPUT
	20	OUTPUT
	40	UPDATE
	80	BACKWARDS
3	01	BUFFERED
	02	UNBUFFERED
	04	KEYED
	08	EXCLUSIVE
	10	PRINT
	20	AXES
4		RESERVED

NDEM - Open conflict mask

This is a mask generated by the compiler containing bits for all attributes which conflict with those on the OPEN statement.

Ordered Delete List (ODL)

Function

Hold list of transient modules to be deleted during program termination.

When Generated

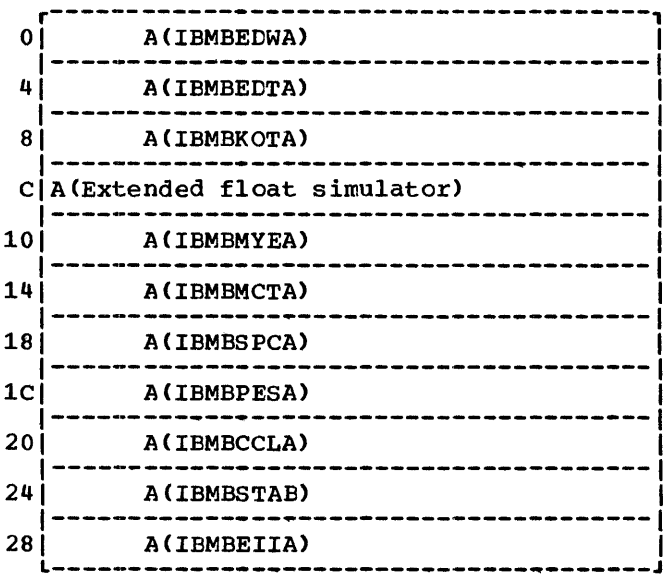
During program initialization.

Where Held

Program Management area.

How Addressed

From offset X'38' in the TCA.



This block is initialized to binary zeros; each routine places its address in the appropriate field as soon as it is loaded.

PLIMAIN

Function

Holds address of entry point of main procedure

When Generated

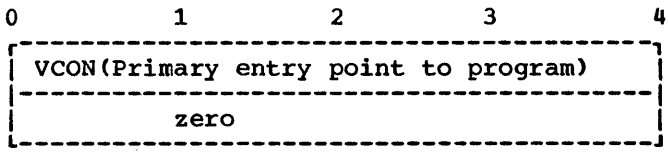
During compilation of procedures with the MAIN option

Where Held

A separate control section in the load module

How Addressed

Address resolved by linkage editor.



Dummy PLIMAIN

A control section in IBMBPIR and IBMTPIR holding addresses of error message module. This control section is link-edited if no compiler generated PLIMAIN exists.

Record Descriptor (RD)

Function

To hold data about the record variable.

When Generated

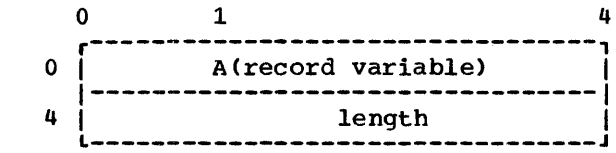
During Compilation.

Where Held

Static control section.

How Addressed

From an offset from register 3 known to compiled code.



Word 1

1. Address of the data to be written out.
2. Address of where data read in is to be put.
3. LOCATE statement: Address of where to store buffer address.

Word 2

Bits 0 - 7 indicate the type of INTO or FROM argument as follows:

X'00' for fixed length strings
X'01' for area variables
X'02' for varying length character strings
X'03' for varying length bit strings

Bits 8-31 length of data to be transmitted (length of variable or buffer for locate mode).

The value is in bytes for all strings including bit strings.

For VARYING strings, the value includes the two length bytes, and is the current length for output operations and the maximum length for input operations.

Request Control Block (RCB)

Function

Used by the record I/O interface module (IBMBRIO) to check the validity of an I/O statement. The instruction in RTMI is carried out by IBMBRIO.

When Generated

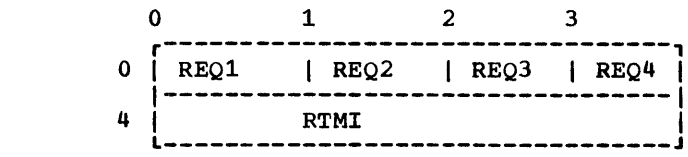
During compilation.

Where Held

Static internal control section.

How Addressed

From an offset from register 3 known to compiled code.



REQ1 (statement identification)

X'00' - READ
X'04' - REWRITE
X'08' - WRITE
X'0C' - LOCATE
X'10' - DELETE
X'14' - UNLOCK

REQ2 (options)

X'80' - INTO/FROM
X'40' - SET
X'20' - IGNORE
X'02' - NOLOCK
X'01' - EVENT

REQ3 (options)

X'80' - KEY
X'40' - KEYTO
X'20' - KEYFROM

REQ4 unused

RTMI

Either a TM or a BR instruction depending on source program.

A TM instruction is used if the statement cannot be checked for validity during compilation, or if it has been checked and found to be invalid.

TM instruction used by IBMBRIO for testing the validity of a statement.

X'91MM2SSS'

where MM is byte containing current statement bit and SSS is offset of corresponding byte in FCB statement mask.

A BR instruction is used if the statement has been checked during compilation and found to be valid.

Unconditional branch instruction to PL/I library or LIOCS transmitter.

Statement Frequency Count Table

Function

To retain a record of the number of times a statement has been branched to or from, for use by the COUNT option.

When Generated

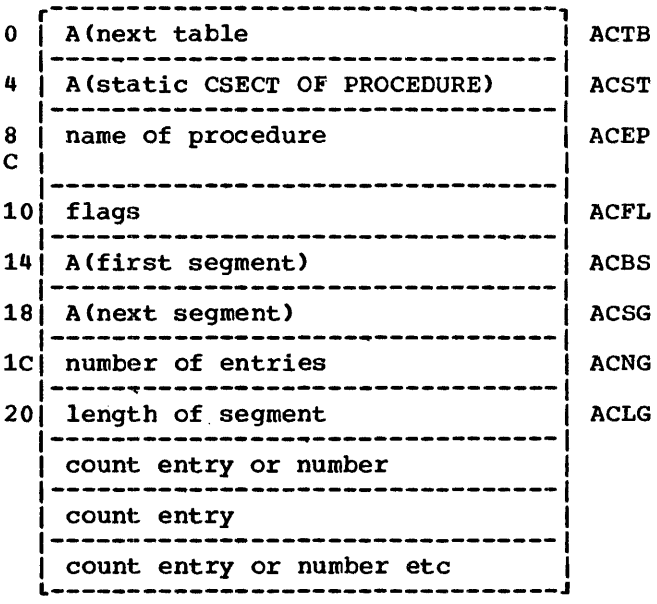
When the associated external procedure is entered.

Where Held

Non-LIFO storage.

How Addressed

The statement frequency count table for the first external procedure in a program is addressed from offset X'48' in the TCA appendage (TIA). The tables are chained together and the chain field of the last table set to zero. The chain field is at offset 0 in the table. The most recently used table is addressed from X'4C' in the TIA.



ACBS The address held in ACBS is the address of ACGS. If tables are segmented, second and subsequent sections of the table will start at a point equivalent to ACSG.

ACFL Flags

ACBI	Bit 0	last update was for a branch in
ACGT	Bit 1	last update was for a GOTO out of block
ACIA	Bit 2	table inactive
ACNM	Bit 3	table is for procedure with GONUMBER option
ACUI	Bit 4	table is uninitialized
ACZL	Bit 5	table contains unexecuted ranges

Other bits unused.

Statement Number Table

Function

To relate statement numbers to offsets so that statement numbers may be given in execution-time messages.

When Generated

During compilation, if the GOSTMT option is in effect.

Where Held

Static internal control section.

How Addressed

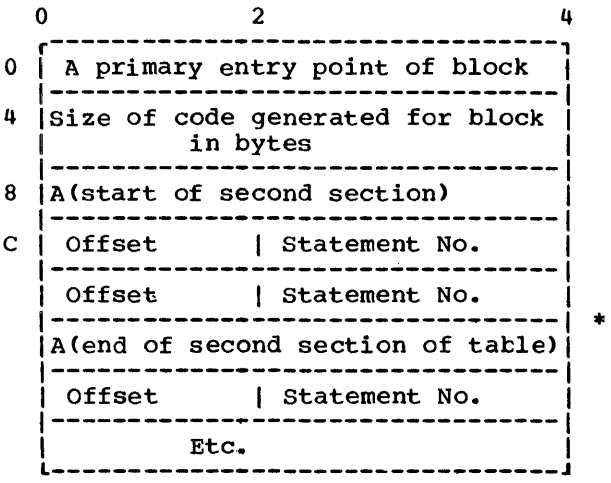
From offset X'8' from entry point of main procedure.

Sections of Table

Because offsets are held in two bytes and the value may in fact take up to three bytes (4096), it is necessary to hold the table in sections.

Statement Number Format

Halfword binary right-aligned.



Line Number Format

When line numbers are generated they are held in 6-byte fields. The first 27 bits hold the line number, right adjusted in binary. The last five bits hold the number of the statement on the line, again right adjusted in binary.

The presence of line numbers is indicated by bit 5 of Flags 2 in the DSA being set to 1. The validity of Flags 2 is indicated by bit 15 in the flags in the first two bytes of the DSA being set to 1. The presence of line numbers is indicated if both these flags are set to 1.

* = End of first section

Offset: Offset is the offset of the first byte of the statement relative to the address of the primary entry point of the block. If the offset is more than X'7FFF' the statement number will be held in the seccnd or subsequent sections of the table. Obtain the number given by translating the offset into binary and ignoring the last 15 bits and step down this number of sections of the table. (For example, if the offset was X'8FFF', translate to binary = '1000 1111 1111 1111'B, ignore last 15 binary digits =1, therefore step down one section of the table. If the offset was X'18FFF' the binary would be '0001 1000 1111 1111 1111'B. Ignoring the 15 right hand bits leaves '11'B therefore step down three sections of the table.)

The address of the second section of the table is held at offset X'8' in the table, the address of the third section is held at the head of the second section, the address of the fourth section at the head of the second section and so forth.

Storage Report Table

Function

To hold the information from which a storage report will be generated.

When Generated

During program or task initialization

Where Held

Program management area, or for major task in storage associated with the control task.

How Addressed

From X'38' in the TIA.

Non-multitasking and PL/I task table

0	True EOS value	TRES
4	Used ISASIZE	TRUS
8	TRFG flags Specified ISASIZE	TRSS
C	ISA adjustment	TRUN
10	Extra storage required	TREX
14	Number of GETMAINs	TRGM
18	Number of FREEMAINs	TRFM
1C	Number of get non-LIFO requests	TRGN
20	Number of free non-LIFO requests	TRFN
24	Current extra storage owned	TRCS
28	Current unused ISA	TRUI
2C	Address of tasking appendage (multitasking only)	TRTT

TRFG_FLAGS

'10000000'B Major task table

'01000000'B Update complete (get LIFO)

Control task table

0	Major task - Used ISASIZE	CSMU
4	Major task - Specified ISASIZE	CSMI
8	Major task - ISA adjustment	CSMN
C	Major task - Extra storage required	CSMX
10	Major task - Number of GETMAINs	CSMG
14	Major task - Number of FREEMAINs	CSMF
18	Major task - Number of get non-LIFO requests	CSMH
1C	Major task - Number of free non-LIFO requests	CSMJ
20	Subtasks - Max ISASIZE used by any subtask	CSXU
24	Subtasks - Min ISASIZE used by any subtask	CSNU
28	Subtasks - Specified ISASIZE all subtasks	CSSI
2C	Subtasks - Max storage required any subtask	CSXN
30	Subtasks - Min storage required any subtask	CSNN
34	Subtasks - Max extra storage any subtasks	CSXX
38	Subtasks - Min extra storage any subtasks	CSNX
3C	Subtasks - Total number of GETMAINS all subtasks	CSSG
40	Subtasks - Total number of FREEMAINs all subtasks	CSSF
44	Subtasks - Total number of get non-LIFO requests all subtasks	CSSH
48	Subtasks - Total number of free non-LIFO requests all subtasks	CSSJ
4C	Maximum number of PL/I tasks attached	CSNA

Stream I/O Control Block (SIOCB)

Function

Holds addresses of source and target, source and target DEDs etc and is used as parameter list by stream I/O routines.

When Generated

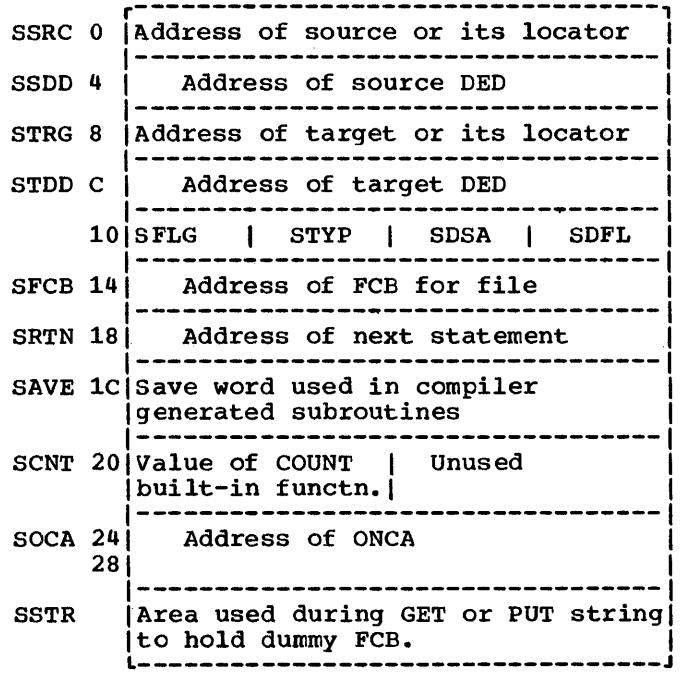
During execution for the duration of the stream I/O statement.

Where Held

In temporary storage.

How Addressed

Passed as parameter list by compiled code.



Flag Byte SFLG

Bit 0 = 1 Transmit on input

Bit 1 = 1 VDA used in edit-directed input

Bit 2 = 1 IBMBSED is used

Bit 3 = 1 Call to IBMSIST required after dealing with next item (Stream I/O only)

SDSA

DSA level number (used only for data-directed I/O)

Type code STYP

Bit 0 = 1 data-directed I/O

Bit 1 = 1 list-directed I/O

Bit 2 = 1 edit-directed I/O

Bit 3 = 1 string I/O

Bit 4 = 1 CHECK entry to data-directed I/O

Bit 5 = 1 input

Data-directed flag SDFL

Bit 0 = 1 Terminating call to data-directed output

String Locator/Descriptor

Function

Used to pass the address and the length of strings to other routines. Also for handling strings with adjustable lengths (e.g., DCL STRING CHAR (N)).

When Generated

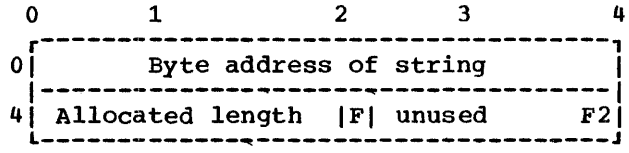
Storage reserved during compilation. Fields completed during execution if string has adjustable length.

Where Held

Static internal control section.

How Addressed

From an offset from register, 3 known to compiled code.



F = '0' B Fixed string (First bit of second byte)
'1' B Varying string

F2 Used for bit strings to hold offset from byte address of first bit in string (3 bits)

Allocated length

For varying strings this is the declared length. Length is held in bits for bit strings and in bytes for character strings.

String Descriptor

The string descriptor is the second word of the string locator/descriptor. It appears in structure descriptors and in the description field of controlled variables.

Structure Descriptor

Function

Contains information about the offset of each element within a structure, and the nature of each element. Used when passing a structure to another routine, or for accessing structure elements during execution, if the structure is declared with adjustable extents or with the REFER option.

When Generated

If the structure has no adjustable elements, during compilation. If the structure has adjustable elements, during execution from information held in the aggregate descriptor descriptor.

Where Held

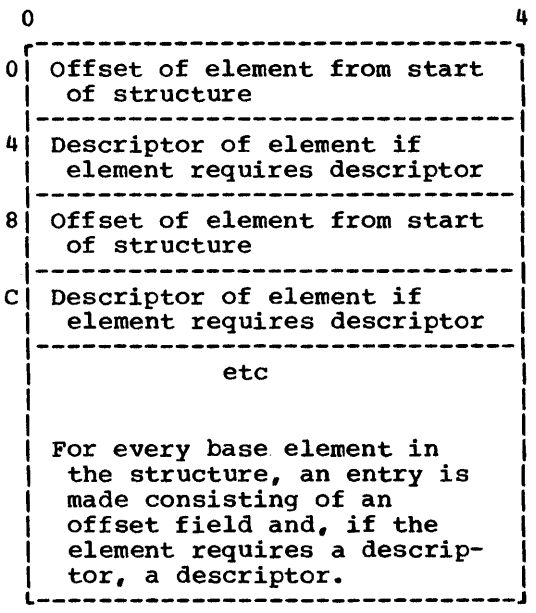
Static internal control section.

How Addressed

From an offset from register 3 known to compiled code.

General Format

For each base element in the structure, a fullword field containing the offset of the start of the element from the start of the structure is given. If the base element is a string, area, or array, this fullword is followed by the offset field for the next base element.



Offset

The offset field is held in bytes, Any adjustments needed for bit-aligned addresses are held in the respective descriptors.

Symbol Table (SYMTAB)

Function

Holds the name of the variable during execution and associates it with the address of the variable. Used only when data-directed I/O or the CHECK condition is specified.

When Generated

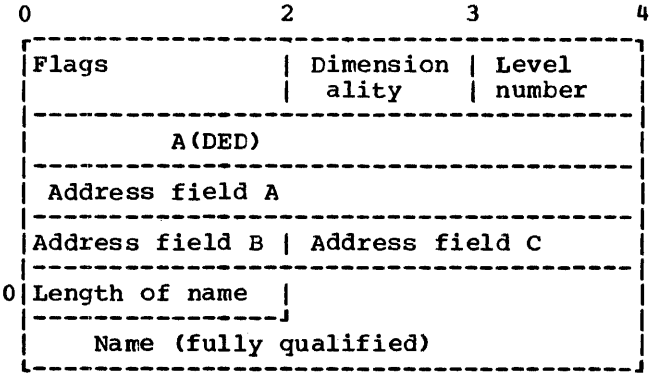
During compilation, if data-directed I/O or the CHECK condition is used in the program

Where Held

Static internal control section for internal names. Separate control section for external names. External control sections consist of the name followed by an *.

How Addressed

From an offset from register 3 for internal data, by an address generated by the linkage editor for external data.



Flags

Bits 0,1 & 2 = '000'B STATIC
= '100'B AUTOMATIC
= '010'B CONTROLLED (not param.)
= '001'B BASED
= '011'B DEFINED
= '101'B a non-CONTROLLED parameter
= '111'B a CONTROLLED parameter

Bit 3 = '1'B EXTERNAL
= '0'B INTERNAL

Bit 4 = '1'B item may appear in some CHECK list
= '0'B item appears in no CHECK list

(Bit 4 must be '1'B if item is EXTERNAL).

Bit 5 = '1'B Address field A refers to data
= '0'B Address field A refers to locator

(Bit 5 must be '0'B for a CONTROLLED parameter)

Bit 6 = '1'B a member of a structure
= '0'B not a member of a structure

Bit 7 = '1'B Normal SYMTAB
= '0'B Short SYMTAB (has fields A, B and C omitted)

Bit 8 = '1'B Address field A addresses code
= '0'B Address field A does not address code

Bit 10 Always set to 0

Bits 11 - 15 reserved: must be set to '0'B.

Dimensionality

The number of dimensions declared for an array item. Dimensionality is zero for other items.

Level number

(for AUTOMATIC, DEFINED, and BASED items. Also for all parameters.) The level of the block in which the variable is declared. The level of a block is one greater than the level of the immediately containing block; the level of the external block is 0.

Address Fields

Addresses are held in different formats for different data types. As far as possible, addresses are held in address field A. However, more information than can be held in a fullword field is sometimes required. When this is the case, address fields B and C are used.

Address field A

If STATIC Address of data or address of locator for items that have locators.

If AUTOMATIC Offset within the associated DSA of the data or of the locator for items that have locators.

If CONTROLLED Offset of the data or its locator from the address in the anchor word.

If BASED Offset of field within DSA containing address of declared pointer qualifier.

If PARAMETER or DEFINED Offset of one word field in associated DSA containing address of corresponding argument, or DEFINED data, or its locator. For CONTROLLED parameters, the argument is its anchor word.

Address field B Used for CONTROLLED and BASED items only.

If CONTROLLED Address of anchor word, either in static internal for internal data or in a separate CSECT for external data.

If BASED See below.

Other data Not used for other data types. Set to a null value of all zeros.

Address field C Used for BASED and structure elements only.

If STRUCTURE (Not BASED structure) Offset from start of structure descriptor to field that holds offset of element from start of structure. See "Structure Descriptor."

If BASED STRING, BASED STRUCTURE, BASED ARRAY, or BASED AREA, For all items except structures, fields B and C hold the offset (right justified) of the descriptor from the start of the DSA in which it is held. For structured items, the offset is to the offset word in the structure descriptor. This word holds the offset of the item from the start of the structure. See "Structure Descriptor".

Other data Not used for other data types. Set to a null value of all zeros.

Length Length is the number of characters in the fully qualified name.

Symbol Table Vector

Function

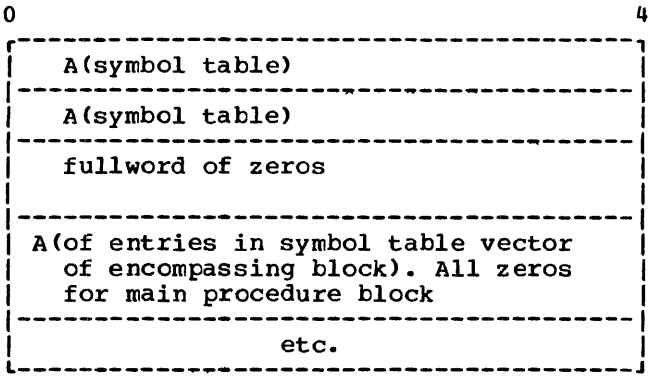
Holds addresses of symbol tables and associates them with the block in which the associated names were declared.

When Generated

During compilation.

Where Held

Static internal control section.



How Addressed

From an offset from register 3 known to compiled code.

General Format

The format of symbol table vector is a series of fullwords. These contain either:

1. The address of a symbol table

or

2. The address of the entry in the symbol table vector of the start of the entries for the encompassing block.

or

3. A fullword of zeros indicating the end of the current block.

Task Communications Area (TCA)

Function

Acts as a central communications area for the program. Contains addresses of essential routines and control blocks, and various flags. (See chapter 5).

When Generated

During program initialization by IBMBPIR.

Where Held

In the program management area at the head of the initial segment area (ISA).

How Addressed

From Register 12

	0	1	2	3	4	
0	TFB0	TFB1	TFB2	TFB3		
4	A(PRV)					TPRV
8	Segment #	BOS Beginning of Segment Pointer				TBOS
C	Segment #	EOS End of Segment Pointer				TEOS
10	Unused					
14	A(current event variable)					
18	A(External Save Area)					TESA
1C	A(TRT Table)					TTRT
20	Task Level					TTIC
24	A(Current Task Variable)					TTSK
28	A(TCA appendage)					TTIA
2C	A(Tasking Appendage)					TTTA
30	A(Save Area for Overflow Routine)					TPSA
34	Open File Chain Anchor					TFOP
38	A(Ordered Delete List)					TODL
3C	Unused					TBUG
40	A(Diagnostic File Block)					TDFB
TORC 44	PL/I Return Code	User Return Code				TURC
48	A(Overflow Routine for Get VDA)					TOVV
4C	A(Flow stmt number table)					TSFT
50	A(Tab table)					TTAB
54	A(Flow module)					TEFL
58	A(LPA Module - Region)					TPSR
5C	A(LPA Module - LPA)					TPSL
60	A(LPA Module - LPA)					TPSM
64	PRV Initialization Word					TPRI
68	Unused					
6C	A(Get Dynamic Storage Routine)					TGET
70	A(Free Dynamic Storage Routine)					TFRE
74	A(Overflow Routine for Get DSA)					TOVF
78	A(Error Handler)					TERR
7C	Environment Description					TENV
80	Normal GOTO Code Used when GOTO out of block may occur					TGTC
F4	A(Interpretive GOTO routine)					TGTM
F8	A(Get control routine)					TGCL
FC	A(Free control routine)					TRCL
100	Dummy ENQ routine field					TEQR
104	Dummy DEQ routine field					TDQR
108	A(WAIT routine)					TAWT
10C	A(COMPLETION pseudovariable routine)					TACP
110	A(EVENT assign routine)					TAEA
114	A(Priority routine)					TAPR
118	A(ENQ/DEQ routines)					TEDR
11C	Reserved for users					TUSR

TENV Is a field used to define PL/I library modification level.

TRLR is the resident library release number.

TTLR is the transient library release number.

TUSR Is a field reserved for the use of programmers using the PL/I Optimizing and Checkout compilers. Any user routine may use this field as a base for addressing.

TFLG contains flag bytes TFB0, TFB1, TFB2, and TFB3.

Flag_Byte_0 - TFB0

TTIS Bit 0 = 0 Major Task 1 Subtask

TTTT Bit 1 = 0 Program will not multitask
1 Program may multitask

TTCK Bit 2 = Reserved

TTFT Bit 3 = 0 Not eldest task from attaching DSA 1 Eldest task

TTFD Bit 4 = 0 No daughter tasks exist 1 Daughter tasks exist

Note: This flag byte is the only one in the TCA used by the central task without synchronising with the subtask. The subtask must never change it. This prevents interference between CPU's on a multiprocessing machine.

Flag_Byte_1 - TFB1

TGFD Bit 0 = 0 No daughter tasks 1 At least one daughter task may exist

TGFE Bit 1 = 0 No active EVENT I/O ON units 1 At least one active EVENT I/O ON unit

TGFV Bit 2 = unused

TGFS Bit 3 = 0 SORT routine inactive 1 SORT routine active

TGNQ Bit 4 = 0 SYSPRINT not ENQed 1 SYSPRINT ENQed by this task

Bit 5 = 1 Task ending

Flag_Byte_2 - TFB2

THQS Bit 0 = 0 Do not raise SIZE for fixed-point divide, fixed-point overflow, exponent overflow, or decimal overflow

1 Raise SIZE if one of these exceptions occurs

THQI Bit 1 = 0 Do not ignore fixed-point divide, fixed-point overflow or exponent overflow 1 Ignore any of these exceptions

Bits 2-4 Reserved

Bit 5 = 1 File associated with SIZE

THQR Bit 6 = 0 Normal action on normal return from on-unit 1 Return to caller after normal return from on-unit

THQC Bit 7 = 0 Not I/O Conversion 1 I/O Conversion

Flag_Byte_3 - TFB3

TMDF Bit 0 = Reserved

Bit 1 = 0 Prompt required
= 1 Prompt not required
Bit 2 = Reserved
Bit 3 = Reserved
Bit 4 = Reserved
Bit 5 = 0 Not implied SKIP next
= 1 Implied SKIP next
Bits 6-7 Reserved

TCA Appendage (TIA)

Function

To hold control and communication information.

When Generated

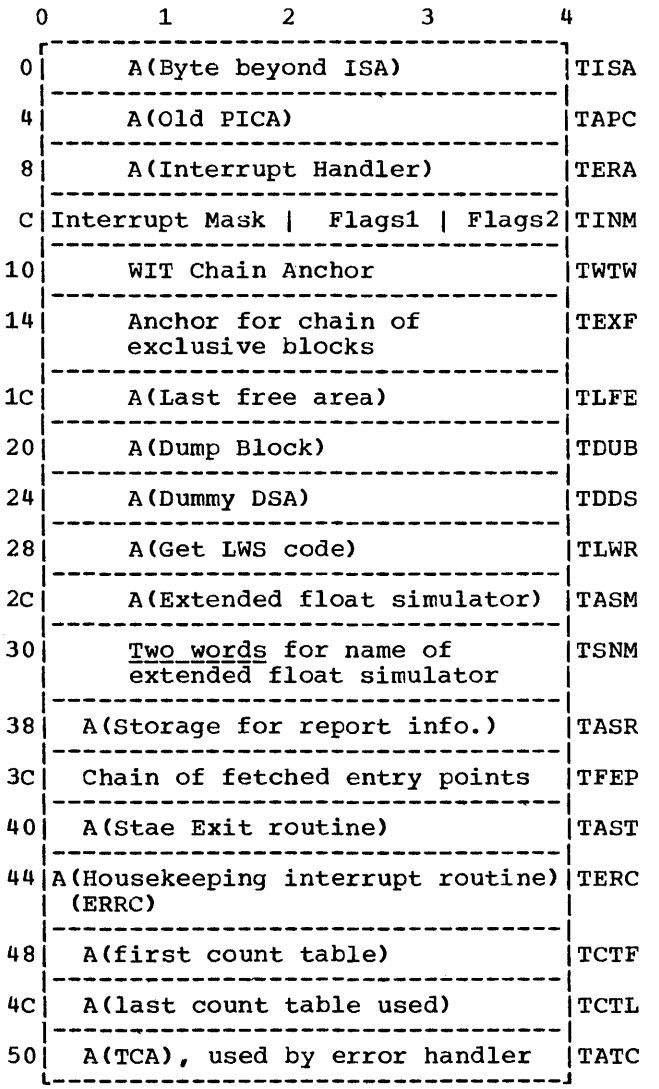
During program initialization.

Where Held

Program management area. Addressed from offset X'28' in the TCA.

How Addressed

From X'28' in the TCA.



Flags1 - TFL1

```
TFLA  Bit 0 = 1 Task terminated normally
TFLS  Bit 1 = 1 SYSPRINT open
TFLJ  Bit 2 = 1 STAE exit in progress
TFLK  Bit 3 = 1 Dump I/O in progress
```

Flags2

```
TFLD  Bit 0 = '1'B caller provided ISA
TFLR  Bit 1 = '1'B storage report required
TFLT  Bit 2 = '1'B  STAE required
TFLP  Bit 3 = '1'B  SPIE required
TFLX  Bit 4 = '1'B Syntax error in program
                   management options
```

TCA Tasking Appendage (TTA)

Function

To hold control and communication information used in multitasking programs.

When Generated

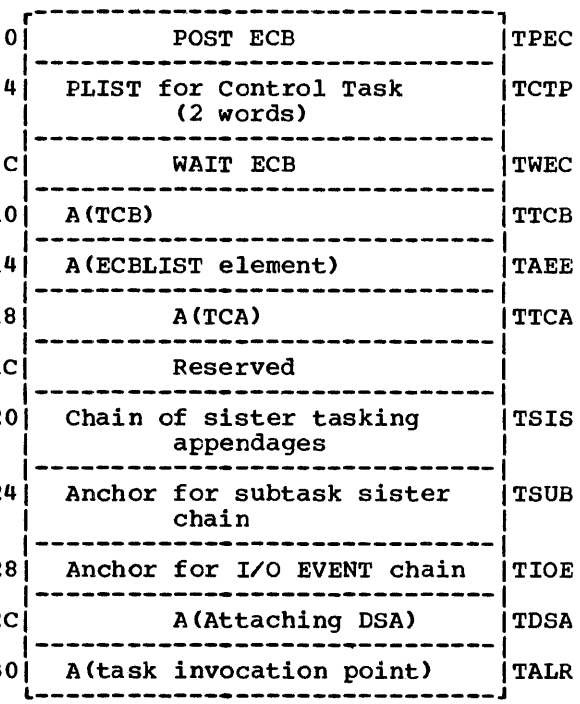
During program initialization.

Where Held

Program Management area.

How Addressed

From X'2C' in the TCA.



Post Codes to Control Task

0	Completion pseudovariable
4	EVENT assignment
8	PRIORITY pseudovariable
12	I/O EVENT completion
16	WAIT termination
20	Detach this block
24	Dedicate control task
28	Liberate control task

Task Variable (TV)

Function

To hold information about task

When Generated

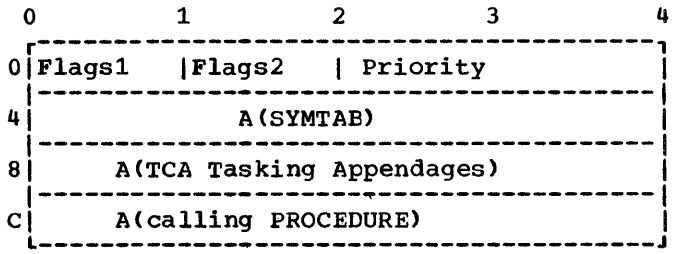
Depends on storage class

Where Held

Depends on storage class

How Addressed

From offset X'24' in the TCA.



Flags1

Bit 0 = 0 Inactive 1 Active

Flags2

Bit 0 = 0 Not a dummy
1 Dummy

Bit 1 = 0 Symbol table does not exist 1
Symbol table exists

Wait Information Table (WIT)

Function

Used to hold information about a WAIT statement

When Generated

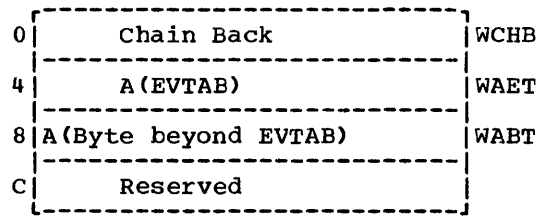
When the WAIT statement is initiated

Where Held

In the LIFO stack

How Addressed

From X'10' in the TIA.



Zygo-lingual Control List (ZCTL)

Function

To hold information required for interlanguage calls. Holds information that does not change for every invocation.

When Generated

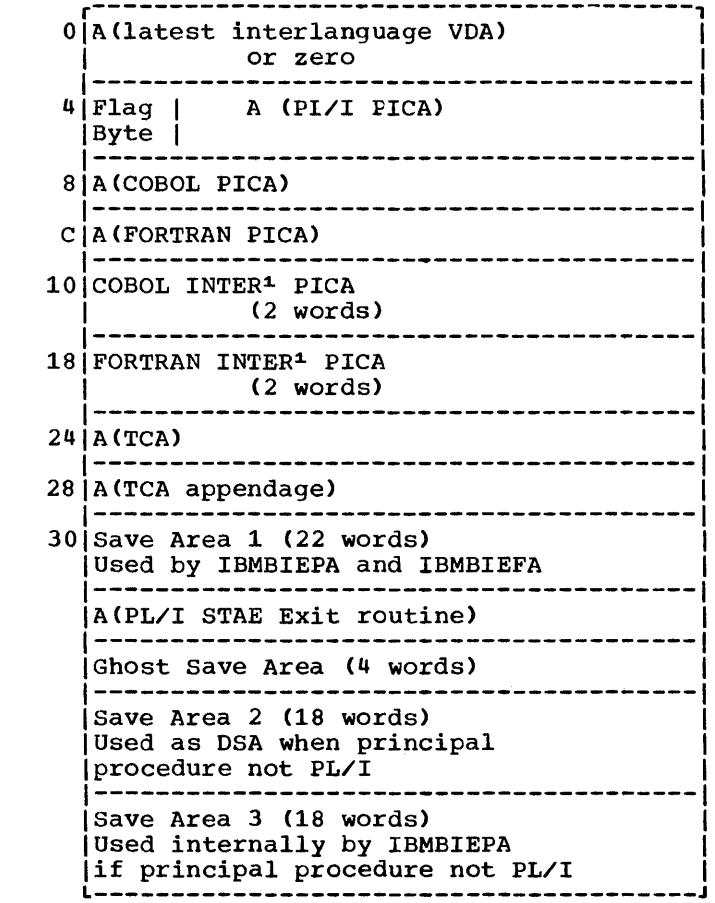
On the first interlanguage call.

Where Held

In the LIFO stack if PL/I is main procedure. If COBOL or FORTRAN are principal procedures, at the head of the unused portion of the region immediately before the TCA.

How Addressed

From offset X'0' in IBMBILCI.



1 The "INTER PICA" is a PICA used if the INTER option is specified.

Note: Beyond offset X'30' only the save areas needed are acquired.

When the first call is made from PL/I to COBOL, only the first 30 bytes are acquired. the first 30 bytes plus save area are acquired.

When the first call is made from COBOL or FORTRAN to PL/I, the complete area shown is acquired.

Flags

Bit 0 = 1 If there is a previous call to COBOL
1 = 1 If there is a previous call to FORTRAN

6 = 1 STAEs will be issued 0 STAEs not issued
7 = 1 SPIEs will be issued 0 SPIEs not issued

Bits 3, 4 and 5 unused

abnormal GOTO 30
  code in TCA 70
  library subroutine IBMBPGO 30
abnormal termination (multitasking) 266
access method
  record I/O 127
  stream I/O 153
activating blocks 24
actual origin (AO) 53
address constants 17
addressing beyond 4K limit 21
aggregates 23
  (see also structure; array)
  address 53
  arrays of structures 23
  COBOL 250
  descriptor descriptor 57,274
  FORTRAN 250
  interlanguage arguments 233,250
  locator 54-56,275
alignment in structures 250
alignment requirements 18
ALL built-in function 191
allocation of storage 75-85
AND logical operation 191
ANY built-in function 191
AO (actual origin) 53
areas
  address 54-56
  control block 273
  descriptor 272
  locator/descriptor 54-56,272
  storage management 84
arguments lists 30
arrays
  assignments 23
  descriptor 276
  descriptors 23
  FORTRAN 250
  implementation of 23
  in stream I/O 174
  interlanguage communication 233
  interleaved 190
  of structures 23,54-56
arrays of structures 23
  arrays 23
  assignments 23
  implementation of 23
ASSEMBLER - PL/I communication 251
ASSEMBLER option 251
attaching a task 264
attributes, data 53
AUTOMATIC variables 19
  in dump 226
  storage 76
backchains
  dynamic 26
  static 26
base element 54-56
base registers
  DSA pointer 17
  program base 17
  static base 17
  TCA pointer 17
BASED storage 20,75
BASED variables 20
  in dump 226
  storage 75
beginning of segment (BOS) pointer 69
BIT data
  string assignment subroutine
   (IBMBBGF) 191
  unaligned strings 189
block enable cells 100
blocks
  activation 24
  termination 24
BOOL built-in function 191
BOS (beginning of segment) pointer 69
bounds, adjustable 53
branches, rationalization of 38
buffer pointers (stream I/O) 156
built-in functions
  arithmetic 189
  array handling 189
  condition 97
  DATE 192
  library subroutines 189
  mathematical 189
  stream I/O 174
  string handling 189
  structure handling 189
  TIME 191

C format item DED 281
CALL statements 26
CALL...TASK failure 266
calling trace
  following through dump 219
  obtaining 210
chain, free-area 71
CHECK condition 105
CHECK prefix 105
checkpoint/restart facility 197
CLOSE statement 143,127
closing files
  explicit closing 143,127
  implicit closing 143
  library subroutines 143
COBOL
  COBOL-PL/I communication 242-246
  option in ENVIRONMENT attribute 251
  structure mapping 250

COBOL (continued)
ZERODIVIDE on-unit 246
COLUMN format option 171
common expression, elimination 34
commoning 36,38
communication
between languages 231-253
between routines 53-66
compare-aligned-bit-string subroutine (IBMBBBC) 191
compare-unaligned-bit-strings subroutine (IBMBBGC) 191
compilation 1
compile-time DED 57
compiler generated subroutines 33
compiler options
AGGREGATE 12
ESD 12
FLOW 12
LIST 12
MAP 12
OFFSET 12
SOURCE 12
STORAGE 12
compiler output 11-38
compiler-generated subroutines 168,179
COMPLETION built-in function 198
COMPLETION pseudovariable 198
completion values, multitasking 268
concatenate-character-strings subroutine (IBMBBCK) 191
CONDITION condition 107
conditions
default values 97
defaults 89-90
enablement 89-90,105
general 97
implementation in general 93-108,93
name abbreviations in dump 210
record I/O 145,144
values in dump 219
consecutive buffered files 127,146
constants 18
constants pool 18
control blocks
formats 271-329
locating in dump 227
control format items 168
DED 280,281
control sections 11-38
control task
general 255
controlled variable block 20
CONTROLLED variables 20
control block 277
header information 20
conversational files 177
conversion 181-189
CONVERSION condition 188
hybrid 187
in-line conversions 185
library subroutines 181-182
multiple 187
ONCA 175
ONCHAR function/pseudovariable 175
ONSOURCE function/pseudovariable 175
stream I/O 160
CONVERSION condition 174,188
COUNT function 174
COUNT option 120
CSECT (control section) 11-38
current enable cell 99

data
internal representation 181
interrupt 88
data element descriptor (DED)
formats 278-281
general description 57
data format item 168
data interrupt 88
data list matching 169
data set interchange between PL/I and COBOL 251
data-directed I/O 168-169
DATAFIELD built-in function 174
DATE built-in function 192
DCLCB see declare control block
decimal overflow interrupt 103
declare control block (DCLCB)
format 282
general 125,127
DELAY statement 192
dequeuing on SYSPRINT 270
descriptors 53-62
aggregate descriptor descriptor 54-56,274
area 54-56,272
array 54-56,276
string 54-56,318
structure 54-56,319
detatching a task 266
DFB (diagnostic file block) 107,283
diagnostic file block (DFB) 108,283
diagnostic statement table (DST) see statement number table
director routines in stream I/O 160
list of 179
disablement of conditions 99
DISPLAY statement 192
DO loops 32
DST (diagnostic statement table) see statement number table
DUB (dump block) 286
dummy arguments in interlanguage communication 250
dummy DSA 72
dummy FCB 22
dummy ONCA
chaining 97
description 71
format 307
introduction 71
dummy sections 21
dump block (DUB) 286
dump control module (IBMBKMR) 109
dump debugging procedures 212,215
dumps
contents 210
debugging with 205-228
file 112
implementation 109-112
obtaining 207
options 207
subroutines that generate 109

dynamic backchain 26
dynamic descendency 87
dynamic ONCB 100
dynamic storage area (DSA) 75-85
  associating DSA with block 219
  contents for compiled code DSA 25
  dummy 72
  format and function 284
  forward chain in dump 224
  IBMBERR's DSA in dump 213
  initialization 65
  uses 75

E format item DED 280
ECB list 264
edit-directed I/O 169-174
  arrays 174
  compiler-generated subroutines 179
  control format items 168
  data format items 168
  FED 168
  format DED 168
  format list 168
  format option handling 173
  GET statement 168
  library director modules 178
  matching data and format lists 173
  non-matching data and format lists 173
  PUT statement 168
element, base 53
element, structure 53
elimination of unreachable statements 36
enable cells 99
enablement of conditions 99
enablement status 28
end of extent, offset to (OEE) 85
end of file 145
end of segment (EOS) pointer 69
END statement 27
ENDFILE condition
  detecting 96
  record I/O 145
  stream I/O 174
  summary information 88
ENDPAGE condition 88
enqueuing on SYSPRINT (reason for) 270
ENTRY data control block 287
entry points
  addresses in dump 218
  conversion subroutines 182
  library subroutines 40-50
  load module 11
  main procedure 11
ENTRY statement in interlanguage calls 231
environment
  definition 3
  FORTRAN 233
  initialization 65-69
  interlanguage communication 238
  SORT 192
ENVIRONMENT attribute COBOL option 251
environment block (ENVB)
  format 288
  record I/O 128
  stream I/O 154
EOS (end of segment) pointer 69
epilogue code 24
epilogue code (continued)
  example 26
error codes, list of 211
ERROR condition 88,211
  on-unit and dumps 215
error handling during execution 87-121
  error code 97,211
  error handling subroutine
   IBMBERR 102,107
  error message modules 107
  event I/O 140
  finding the block entry-point
   address 107
  FORTRAN 247
  record I/O 145
  stream I/O 174
error identification
  address in dump 218
  using dump in general 205-228
ESD records
  definition 11
  for conversion modules 182
established on-units 100
event control block (ECB) 256-257,202
event I/O 132-135
event table (EVTAB) 289
event variables 197,290
  control block format and function 290
exclusive block IOCB (XBI) 291
exclusive file block (XBF) 292
exclusive I/O 140
execute interrupt 88
execution time options
  handling 68
  PLIXOPT 68
exit table, SORT 197
explicit open
  stream I/O 153
exponent overflow interrupt 88
exponent underflow interrupt 88
extent, offset to end of (OEE) 85
external conversion director modules 180
EXTERNAL data 65
external reference, weak(WXTRN) 44

F format item DED 280
FCB see file control block
FCBA field in FCB 156
FCPM field in FCB 175
FED (format element descriptor)
  format 280-281
FEFT field in FCB 145
FEMT field in FCB 145
FERM field in FCB 145
fields, locating in dump 227
file control block (FCB) 128
  FCBA field 156
  FCPM field 175
  FEFT field 145
  FEMT field 145
  FERM field 145
  fields for buffer operation 156
  format and function 293-296
  FREM field 156
  general description 128
  record I/O 295
  stream I/O section 296

filenames 127
files
  addressing 21
  closing 143,127
  conversational 177-178
  declaration 125,127
  declaration with COBOL option 251
  exclusive 143
  filename 127
  implicit opening 144,127
  information in dump 211
  opening, explicit 128
  record variable 130
FINISH condition 69,90
fixed-point data
  binary 181
  decimal 181
  DED 279
FIXEDOVERFLOW condition 88
floating-point data
  binary 181
  decimal 181,185
  divide interrupt 88
  underflow interrupt 103
floating-point registers
  saving 103
  usage 19
FLOW compiler option 118-121
  library subroutine IBMBEFL 114
flow of control 24-30
flow statement table (FST)
  format 297
format element descriptor (FED)
  description 61
  format and function 280
format items 173
format list matching 173
format option handling 171
formatting modules in stream I/O 180
formatting, conversational files 177
FORTRAN interrupt 247
FORTRAN-PL/I communication 246-248
free control routine 266
free decimal 184
free-area chain 79
freeing storage 75-85
FREM field in FCB 156
function references 27
fundamental in-line conversions, list of 185

get control routine 266
GET DATA statement
  implementation in general 160,168-169
  symbol tables and symbol table vectors 61
GET LIST statement 168
GETIME macro instruction 192
GOTO only on-units 30
GOTO statement 27-29
  label variable 29
  out of block 28
  within block 28

hardware interrupts see program check interrupts
hexadecimal dump module (IBMBKDO) 109
hierarchy of tasks 255
hybrid conversion 188

I/O
  record 125-151
  stream 146
  stream conditions, detecting 96
IBMBAAH 191
IBMBAIH 189-190
IBMBAMM 191
IBMBANM 191
IBMBAPC 191
IBMBAPE 191
IBMBAPF 191
IBMBAPM 191
IBMBASC 191
IBMBASE 191
IBMBASF 191
IBMBAYE 191
IBMBAYF 191
IBMBBBA 191
IBMBBBC 191
IBMBBBN 191
IBMBBCI 191
IBMBBCK 191
IBMBBCT 191
IBMBBCV 191
IBMBBGB 191
IBMBBGC 191
IBMBBGF 191
IBMBBGI 191
IBMBBGK 191
IBMBBGS 191
IBMBBGT 191
IBMBBGV 191
IBMBEFL 118-121
IBMBERR
  DSA in dump 213
  general 102-106
IBMBESM 106-109
IBMBESN 106-109
  messages 106-109
IBMBIEC 242-246
IBMBIEF 246-248
IBMBIEP 248-250
IBMBILC1 (interlanguage root control block) 238,298
IBMBJWT 198
IBMBMXE 190
IBMBMXL 190
IBMBMXS 190
IBMBMXW 190
IBMBMXY 190
IBMBMXZ 190
IBMBMYE 190
IBMBMYL 190
IBMBMYS 190
IBMBMYX 190
IBMBMYY 190
IBMBMYZ 190
IBMBOCA 126
IBMBOCL 126
IBMBOPA 126
IBMBOPB 126
IBMBOPC 126

IBMBPAF, controlled variable allocation 20
IBMBPAM 85
IBMBPEP - exceptional error message director 112
IBMBPEQ - NO MAIN PROCEDURE message 112
IBMBPER - NO STORAGE message 112
IBMBPES - ABEND analyzer 112
IBMBPET - abnormal interrupt message 112
IBMBPEV - ABEND analyzer 112
IBMBPGD 82
IBMBPGO, interpretive GOTO 30
IBMBPGR, storage management 79-82Z
IBMBPII 68
IBMBPIR 68
IBMBPSL 44
IBMBPSM 44
IBMBPSR 44
IBMBRAA regional seq output trans 126
IBMBRAB regional seq output trans 126
IBMBRAC regional seq output trans 126
IBMBRAD regional seq output trans 126
IBMBRAE regional seq output trans 126
IBMBRAF regional seq output trans 126
IBMBRAG regional seq output trans 126
IBMBRAH regional seq output trans 126
IBMBRAI regional seq output trans 126
IBMBRBA regional seq in/upd trans 126
IBMBRBC regional seq in/upd trans 126
IBMBRBD regional seq in/upd trans 126
IBMBRBE regional seq in/upd trans 126
IBMBRBF regional seq in/upd trans 126
IBMBRBG regional seq in/upd trans 126
IBMBRCA unbuffered consec trans 126
IBMBRCB unbuffered consec trans 126
IBMBRCC unbuffered consec trans 126
IBMBRDA regional direct non-exc trans 126
IBMBRDB regional direct non-exc trans 126
IBMBRDC regional direct non-exc trans 126
IBMBRDD regional direct non-exc trans 126
IBMBREA record I/O error module 126
IBMBREB record I/O error module 126
IBMBREC record I/O error module 126
IBMBREF ENDFILE module 126
IBMBRIO record I/O interface 126
  description of functions 126
  parameter list 130
IBMBRJA indexed seq in/upd trans 126
IBMBRJB indexed seq in/upd trans 126
IBMBRKA indexed direct no-exc trans 126
IBMBRKA indexed direct non-exc trans 126
IBMBRKB indexed direct non-exc trans 126
IBMBRKC indexed direct non-exc trans 126
IBMBRLA indexed direct non-exc trans 126
IBMBRLB indexed direct non-exc trans 126
IBMBRQA buffered consec non-spanned trans 126
IBMBRQB buffered consec non-spanned trans 126
IBMBRQD buffered consec non-spanned trans 126
IBMBRQE buffered consec input spanned trans 126
IBMBRQF buffered consec output spanned trans 126
IBMBRQG buffered consec update spanned trans 126
IBMBRTP teleprocessing input trans 126
IBMBRXA exc regional direct upd/input trans 126
IBMBRXB exc regional direct upd/input trans 126
IBMBRXC exc regional direct upd/input trans 126
IBMBRXD exc regional direct upd/input trans 126
IBMBRYA exc indexed direct upd/input trans 126
IBMBRYB exc indexed direct upd/input trans 126
IBMBRYB exc indexed upd/input trans 126
IBMBRYC exc indexed direct upd/input trans 126
IBMBRYD exc indexed direct upd/input trans 126
IBMBSAI 180
IBMBSAO 180
IBMBSCI 180
IBMBSCO 180
IBMBSCP, COPY module 175,180
IBMBSCV 180
IBMBSDI 179-180
IBMBSDO 179-180
IBMBSED 179-180
IBMBSEI 179-180
IBMBSEO 179-180
IBMBSFI 180
IBMBSFO 180
IBMBSIC 177
IBMBSII 179
IBMBSIO 179
IBMBSIS 180
IBMBSLI 179-180
IBMBSLO 179-180
IBMBSOC 177
IBMBSOF stream output file trans 126,179-180
IBMBSOU stream output file trans 126,179-180
IBMBSOV 179-180
IBMBSPC 177,180
IBMBSPI 180
IBMBSPL 179-180
IBMBSPO 180
IBMBSTAB 171
IBMBSTF stream output print file trans 126,179-180
IBMBSTI stream input trans 126,179-180
IBMBSTU stream output print file trans 126,180
IBMBSTV stream output print file trans 126,179-180
IBMBSXC 180
IBMTJWT wait module multitasking 269
IBMTPIR 260
IBMTPSL 44
IBMTPSR 44
IELCGBB, test for '1' bits 34
IELCGBO, test for '0' bits 33
IELCGCB, compare long bit 33
IELCGCL, compare long 33
IELCGIA, stream input 33
IELCGIB, stream input 33
IELCGMV, move long 33
IELCGOA, stream output 33
IELCGOB, stream output 33

IELCGOC, stream I/O X format items 33,171
IELCGON, dynamic ONCB chaining 33
IELCGRV, revert VDA chaining 33
implicit close in record I/O 143,127
implicit open
  record I/O 127,144
  stream I/O 156
in-line conversion 183-185
  example of 186
  list of fundamental types 185
in-line record I/O 146
INDEX built-in function 191
indexing interleaved arrays 190
initial storage area (ISA) 8,68,75,75-78
initialization
  FORTRAN 238
  PL/I 65-69
  program 65-69
  stream I/O subroutines 178
input/output control block (IOCB) 301
instruction, associating with module 218
INTER option 242
interlanguage communication 231-253
  aggregate arguments 231-252,250
  arrays 250,233
  ASSEMBLER option 251
  basic rules 231
  COBOL option of ENV attribute 251
  control blocks 238
  FORTRAN calls PL/I 248
  IBMBILC1 238
  interrupt handling 247,249
  interrupt in COBOL 246
  interrupt in FORTRAN 247
  interrupt in PL/I 249
  NOMAP option 250
  NOMAPIN option 250
  NOMAPOUT option 250
  PL/I calls COBOL 242
  PL/I calls FORTRAN 246
  principles 233
  storage 250
  structures 250
  use of locators 233
  VDA 238
  ZCTL 238
interlanguage root control block (IBMBILC1) 298
interlanguage VDA 299
internal form of data types 181
interpretive code, need for
interpretive GOTO routine IBMBPGO 30
interrupt control block (ICB) 300
interrupt handling 87-121
  COBOL 246
  event I/O 143
  FORTRAN 247
  library subroutine IBMBERR 102-106
  return from 104
interrupt identification using dump 205-228
  in library module 224
  interrupt address 215
invert-aligned-bit-string subroutine (IBMBBBN) 191
IOCB (input/output control block) 301
ISA (initial storage area) 68,75
  multitasking 85
KEY condition 88
key descriptor (KD) 130,304
key variable 135

label data format 305
label variables
  errors when using 30
  format 305
  general description 29
last free area (TLFE) 71
last-in/first-out (LIFO) storage 75-80
library calls 32
  addressing 32
  example of calling sequences 32
  general 30
  mnemonic letter usage 32
  within TCA 32
library subroutines 31-32
  arithmetic 189
  array handling 189-191
  computational 189
  conversion package 181
  in record I/O 126
  in stream I/O 179
  mathematical 189
  naming conventions 32
  string handling 189
  workspace 42
library workspace (LWS)
  description 42
  format and function 306
LIFO (last-in/first-out) storage 75-79
LINE format option 171
link-editing 65
list-directed I/O 160
listing conventions 15
load module
  entry point 11
LOCATE statement 125,130-132
locators 53-60
  aggregate locator format and function 275
  area locator/descriptor format and function 272
  string locator/descriptor format and function 318
logical operation subroutines 191

main procedure
  no main procedure 65
  termination 69
major free area 76
modification of do-loop control 38
module, object 11
movement of expressions out of loops 34
multiple conversion 188
multitasking 255-270
  completion values 268
  following chaining in dump 225
  housekeeping 256
  ISA acquiring 85
  library 260
  POSTCODES 256
  priority 256
  reading dumps, general 228
  TCA tasking appendage (TTA) 256,326

multitasking (continued)
WAIT statement 268

NAB (next available byte) pointer 76
NAME condition 174,88
naming of library modules 32
next available byte (NAB) pointer 76
NOCHECK prefix 105,106
NOCONVERSION prefix 188
NOMAP option 250
NOMAPIN option 250
NOMAPOUT option 250
non-LIFO storage 76,79
NOSPIE option 68
NOSTAE option 68
null on-unit 102
NULL values 21

object module 11
object program listing, contents 15
object program listing, example 14
OCB see open control block
OCCURS (COBOL) 250
ODL (ordered delete list) 310
OEE (offset to end of extent) 85
offsets null value 21
ON CHECK 106
ON communications area (ONCA)
chain in dump 223
description 97
dummy 97
format and function 307
ON control block (ONCB)
description 100
format and function 308
ON statement 100-102
on-cells 102
on-code 97
on-units 30,102
GOTO only 30
ONCA 72,307
ONCHAR function/pseudovariable 174
ONSOURCE function/pseudovariable 174
open control block (OCB)
format 309
function 128
OPEN statement 128
opening files
explicit open for record I/O 128,125
implicit open for record I/O 127,143
operating system interfaces
miscellaneous 191-203
see also I/O; error handling; initialization
operation interrupt 88
optimization
branching around redundant expressions 37
commoning constants and control blocks 36
effect on conversions 181
elimination of common expression 34
elimination of unreachable statements 36
examples of 33-38
general 33-38

optimization (continued)
modification of do-loop control 38
movement of expressions out of loops 34
rationalization of branches 36
OR logical operation 191
ordered delete list (ODL) 310
output, compiler 11-38
overflow routine, IBMBPGR 79,71

packed intermediate decimal format 181
PAGE format option 171
parameter lists 30
contents in dump 227
PICTURE data
DEDs 280
FEDs 280
PL/I - ASSEMBLER communication 251
PL/I-COBOL communication 231-253
PL/I-FORTRAN communication 231-253
PLIBASE 255
PLICALLA 68
PLICALLB 68
PLICKPT 197
PLICOUNT 120
PLIDUMP facility
how to use 207,215
implementation 109
PLIFLOW 11,120
PLIMAIN 65
format 311
PLISORT 192,197
PLISTART 65,11
PLITABS 171
PLITASK 255
PLIXOPT 68
pointers
BOS 76
COPY option 175
EOS 76
NAB 76
storage handling 76
pointers, null value 21
POLY built-in function 191
POST ECB 256
POSTCODEs list of 256
PRINT files 171
priority of task 256
privileged operation interrupt 88
program check interrupts 96
program control data 22
DED 278-281
program control section 11
contents 18
program management area 69-71
prologue 24
example 24
prompting 177-178
protection interrupt 88
pseudo register vector
general description 21
initialization 22
location of 22
purge task subroutine 266
PUT statement 160

qualified conditions 87

READ statement 130-132,125
recompilation to obtain dump, avoiding 209
RECORD condition 88
record descriptor (RD) 135,312
record I/O 125-151
  control blocks generated 128
  error handling 144
  in line 146
  interface routine (IBMBRIO) 130
  library routines 126
  library-call 125
  VSAM data sets 130
record variable 135
redundant expressions, branching around 37
REFER option 54-56,191
register usage 17
  branch and link 18
  compiled code 18
  DO loop control 19
  preferred registers 19
  static backchain 19
registers 19
  even/odd pairs 19
  floating point 19
  library usage 19
  work 19
relative virtual origin (RVO) 53,54-56
REPEAT built-in function 191
REPLY option 192
report table 82
request control block (RCB)
  description 130
  format and function 313
RETURN statement 27

save areas
  IBMBPGR 71
  IBMBPIR 71
  registers in dump 220
SAVE field in SIOCB 156
SCNT field in SIOCB 156
SFCB field in SIOCB 156
SFLG field in SIOCB 156
shared library 44-50
  initialization 50
  link pack area 48
  multitasking 50
  region 48
SIGNAL statement 96
significance interrupt 96
SIZE condition 88,103
SKIP format option 171
SLD (string locator/descriptor)
  subroutine 191
SOCA field in SIOCB 156
software interrupts
  definition 87
  main discussion 103-105
SORT exit 197
sort/merge facility 192
source definition 146
specification interrupt 88
squashed mode 178
SRTN field in SIOCB 156
SSDD field in SIOCB 156
SSRC field in SIOCB 156
SSTR field in SIOCB 156
standard save area, operating system 75
statement frequency count table
  discussion 120
  format and function 314
statement number
  in messages 107
  of error, in dump 221
statement number table (SNT or DST) 315
  location in dump 227
static backchain 26
  in dump 224
static control sections 15
static descendency 87
static internal control section 15
static ONCBs 100
static storage map 15
  example 13
static variables 21
  locating in dump 226
STATUS function/pseudovariable 197
STDD field in SIOCB 156
storage
  initial storage area (ISA) 68
  main discussion 75-85
  routine IBMBPGR 77-82
  segments 76,79
storage report table
  format 316
storage reports 82-85
  implementation 82-85
  information given 82
  multitasking 84
stream I/O 146
  built-in functions 174
  conditions 174-176
  COPY option 175
  COUNT function 174
  DATAFIELD function 174
  director routines 179
  end of file 174
  external conversion director modules 180
  file opening 156
  format items 169
  format lists 171
  formatting modules 180
  implicit open 156
  initializing modules 179
  ONCHAR function/pseudovariable 174
  ONSOURCE function/pseudovariable 174
  transmitter modules 179-180
stream I/O control block (SIOCB) 156,317
STRG field in SIOCB 156
strings
  adjustable 189
  built in functions 189,191
  DED 278
  descriptor 318
  FED 281
  length 54-56
  locator/descriptor 54-56,318
  STRING function/pseudovariable 191
  STRING option 175
  STRINGRANGE condition 90
  STRINGSIZE condition 90,189
structures
  assignments 23
  COBOL 250

structures (continued)
descriptor 54-56,319
element (definition) 53
implementation of 23
interlanguage communication 250,233
mapping 250,54-56
of arrays 54-56
structures of arrays 23
subroutines, compiler-generated 180
SUBSCRIPTRANGE condition 90
SUBSTR built-in function 191
SUM built-in function 191
symbol table (SYMTAB)
format 320
general 61
symbol table element list see symbol table vector
symbol table vector 61
format 322
SYMTAB see symbol table
system action, standard
definition 87
system detected interrupts 96

tab table 171
target, definition 146
task communications area (TCA)
appendage (TIA) 71,325
description 69-71
format 323
tasking appendage (TTA) 256,326
task variable
format and function 327
tasking appendage (TTA) 256,326
TCA see task communications area
temporaries 20
temporary variables (temporaries) 20
storage 75
termination of program 69
TIA (TCA implementation appendage) 71,325
TIME built-in function 191
trace
FLOW option 114
following through dump 219
information in dump 210
transfer vector 44
transient library 40-50
TRANSLATE built-in function 191
transmission statement
in record I/O 125,130
TRANSMIT condition 145
detecting 96
transmitter modules
record I/O 126
stream I/O 179-180
TSO (time sharing option) 177-178
TTA (TCA tasking appendage)
format 326
general 256-258
TXT records 11

unaligned bit strings 181
UNDEFINEDFILE condition 88
UNDERFLOW condition 88
unqualified conditions 87
use of locators 233
user exits 192

variable data area (VDA)
interlanguage communication 238,299
variables 20
AUTOMATIC 19
BASED 20
CONTROLLED 20
entry 287
EXTERNAL 65
label 305
locating in dump 226-228
pointer 21
STATIC 21
temporaries 20
varying-length strings
internal representation 181
VERIFY built-in function 191
VSAM data sets 130
opening 130

WAIT ECB 256
wait event table (WIT) 268,328
WAIT macro instruction 192
WAIT statement
multitasking 268
non-multitasking 198
weak external reference (WXTRN) 44
WIT (wait event table) 268,328
WRITE statement 130-132,125
WXTRN (weak external reference) 44

X format items 171,180
XBF (exclusive file block) 292
XBI (exclusive block IOCB) 291

ZCTL (zygo-lingual control list) 238,329
ZERODIVIDE condition 88
zygo-lingual control list (ZCTL) 238,329

OS PL/I Optimizing Compiler:
Execution Logic

SC33-0025-2

READER'S COMMENT FORM

Your views about this publication may help improve its usefulness; this form will be sent to the author's department for appropriate action. Using this form to request system assistance or additional publications will delay response, however. *For more direct handling of such requests, please contact your IBM representative or the IBM Branch Office serving your locality.*

Possible topics for comment are:

Clarity Accuracy Completeness Organization Index Figures Examples Legibility

Cut or Fold Along Line

What is your occupation? _____

Number of latest Technical Newsletter (if any) concerning this publication: _____

Please indicate in the space below if you wish a reply.

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments.)

SC33-0025-2

Your comments, please . . .

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. Your comments on the other side of this form will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Cut or Fold Along Line

Fold Fold

First Class
Permit 40
Armonk
New York

Business Reply Mail
No postage stamp necessary if mailed in the U.S.A.

Postage will be paid by:

International Business Machines Corporation
Department 813(HP)
1133 Westchester Avenue
White Plains, New York 10604

Fold Fold

IBM

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

OS PL/I Optimizing Compiler: Execution Logic (File No. S360/S370-29) Printed in U.S.A. SC33-0025-2