V2:PROC(P) OPTIONS(RE...)
ON ERROR
CALL PLITEST;
DCL 字(1000000) BASE
ALLOCATE 字;
CALL PROC(字);
END;
OS PL/I Version 2

Problem Determination

Release 1
First Edition (December 1987)

This edition applies to Version 2 Release 1 of the following licensed programs:

- OS PL/I Optimizing Compiler, Library, and Interactive Test Facility, Licensed Program 5668-909
- OS PL/I Optimizing Compiler and Library, Licensed Program 5668-910
- OS PL/I Library Only, Licensed Program 5668-911

and to any subsequent releases until otherwise indicated in new editions or technical newsletters.

Changes are made periodically to this publication; before using this publication in connection with the operation of IBM systems, consult the latest IBM System/370, 30xx, and 4300 Processors Bibliography, GC20-0001, for the editions that are applicable and current.

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

Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality. If you request publications from the address given below, your order will be delayed because publications are not stocked there.

A form for readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, P.O. Box 49023, Programming Publishing, San Jose, California, U.S.A. 95161-9023. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

This is a licensed document that contains restricted materials of International Business Machines Corporation. © Copyright International Business Machines Corporation 1985, 1987. All rights reserved.
About This Book

Use *OS PL/I Version 2 Problem Determination* with the OS PL/I Optimizing Compiler Version 2. It will help you to determine problems, to formulate software search facility (SSF) search arguments and to document and submit APARs.

You can also use PLITEST, and PLITEST's interactive facilities, during run-time to diagnose problems. However, this book does not discuss using PLITEST for problem determination. In a few instances, where using PLITEST would help you significantly, PLITEST is discussed briefly. If you have PLITEST installed on your system and want more information about using PLITEST, see *OS PL/I Version 2 Programming: Using PLITEST*.

Who Might Use This Book

The primary audience for this book is all application programmers and systems programmers who write and run PL/I programs.

CMS and TSO Considerations

This book often refers to *ddnames*. In CMS, ddnames are associated to files by means of the FILEDEF command. In TSO, ddnames are associated to files by means of the ALLOC command.

How This book Is Organized

- *Chapter 1, Compiler Overview*, briefly describes how a PL/I program is compiled, link edited and run.
- *Chapter 2, Compile-Time Problem Determination*, contains the compile-time problem determination flowchart which describes various problems and recommends solutions for problems that can occur at this time.
- *Chapter 3, Compiler Output*, describes the object module generated by the compiler. It tells how PL/I organizes the object module and the form the information within it takes.
- *Chapter 4, Run-Time Organization*, describes the facilities PL/I uses during run-times such as registers, storage, load modules and the multitasking library.
- *Chapter 5, Run-Time Problem Determination*, contains the run-time problem determination flowchart which describes various problems and recommends solutions for problems that can occur at this time.
- *Chapter 6, Debugging Using Dumps*, describes how to get a PL/I dump and how to interpret its contents.
- *Chapter 7, Using SSF and CSSF Search Arguments*, describes how to formulate arguments for and how to use the Software Search Facility (SSF) or the Customer Software Search Facility (CSSF).
- *Chapter 8, Submitting an APAR*, describes how to document and report suspected product problems to IBM using an APAR (Authorized Program Analysis Report).
Appendix A, Control Blocks, gives information about control blocks that may be useful in problem determination.

Appendix B, Record-Oriented Input/Output, describes record I/O.

Appendix C, Stream-Oriented Input/Output, describes stream I/O.

Other OS PL/I Version 2 Books Available

The complete list of books in the OS PL/I Version 2 library is given in Figure 1. The figure shows what tasks each book is designed to help you with.

<table>
<thead>
<tr>
<th>Task</th>
<th>OS PL/I Version 2 Publications</th>
<th>Order Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluation</td>
<td>General Information</td>
<td>GC26-4313</td>
</tr>
<tr>
<td></td>
<td>Licensed Program Specifications</td>
<td>GC26-4314</td>
</tr>
<tr>
<td>Installation and</td>
<td>Installation and Customization under MVS</td>
<td>SC26-4311</td>
</tr>
<tr>
<td>Customization</td>
<td>Installation and Customization under CMS</td>
<td>SC26-4312</td>
</tr>
<tr>
<td>Application and System</td>
<td>Programming Guide</td>
<td>SC26-4307</td>
</tr>
<tr>
<td>Programming</td>
<td>Language Reference</td>
<td>SC26-4308</td>
</tr>
<tr>
<td></td>
<td>Reference Summary</td>
<td>SX26-3759</td>
</tr>
<tr>
<td></td>
<td>Using PLITEST</td>
<td>SC26-4310</td>
</tr>
<tr>
<td></td>
<td>Messages and Codes</td>
<td>SC26-4309</td>
</tr>
<tr>
<td>Diagnosis</td>
<td>Problem Determination</td>
<td>LY27-9528</td>
</tr>
</tbody>
</table>

Figure 1. The OS PL/I Version 2 Library

The following books are the ones that will be most helpful while you are using this book:


OS PL/I Version 2 Programming: Messages and Codes, SC26-4309, lists error messages and codes that may be issued when you compile, link edit, and run OS PL/I programs. It also includes messages and codes issued by PLITEST. This book lists messages, and where needed, explanations, examples, and suggested programmer response.
Other Books You Might Need

You may want to refer to the following publications during your problem determination process:

OS/VS2 System Programming Library: Debugging Handbook, GC28-0708, GC28-0709, GC28-0710
OS/VS2 System Programming Library: Service Aids, GC28-0647
OS/VS2 System Programming Library: MVS Diagnostic Techniques, GC28-0725
OS/VS2 TSO Command Language Reference, GC28-0646
MVS/370 Message Library: System Messages, GC38-1008
MVS/Extended Architecture Command Language Reference, GC28-0646
MVS/Extended Architecture Message Library: System Messages, GC28-1156
MVS/Extended Architecture Message Library: System Codes, GC28-1157
MVS/Extended Architecture Message Library: TSO Terminal Messages, GC38-1046
MVS/Extended Architecture Diagnostic Techniques, LY28-1199
MVS/Extended Architecture System Programming Library: Service Aids, GC28-1159
TSO EXTENSION Command Language Reference, SC28-1307
Virtual Machine/System Product CMS Command Reference, SC19-6209
## Contents

<table>
<thead>
<tr>
<th>Chapter 1. Compiler Overview</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>1</td>
</tr>
<tr>
<td>Processing a PL/I Program</td>
<td>2</td>
</tr>
<tr>
<td>Compilation</td>
<td>2</td>
</tr>
<tr>
<td>Link Editing</td>
<td>4</td>
</tr>
<tr>
<td>Running</td>
<td>4</td>
</tr>
</tbody>
</table>

| Chapter 2. Compile-Time Problem Determination | 5 |

<table>
<thead>
<tr>
<th>Chapter 3. Compiler Output</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Organization of This Chapter</td>
<td>14</td>
</tr>
<tr>
<td>Listing Conventions</td>
<td>14</td>
</tr>
<tr>
<td>Static-Storage Map</td>
<td>15</td>
</tr>
<tr>
<td>Object-Program Listing</td>
<td>18</td>
</tr>
<tr>
<td>Static Internal Control Section</td>
<td>22</td>
</tr>
<tr>
<td>Program Control Section</td>
<td>23</td>
</tr>
<tr>
<td>Register Usage</td>
<td>24</td>
</tr>
<tr>
<td>Library Register Usage</td>
<td>26</td>
</tr>
<tr>
<td>Handling and Addressing Variables and Temporaries</td>
<td>27</td>
</tr>
<tr>
<td>Automatic Variables</td>
<td>27</td>
</tr>
<tr>
<td>Compiler-Generated Temporaries</td>
<td>27</td>
</tr>
<tr>
<td>Controlled Variables</td>
<td>27</td>
</tr>
<tr>
<td>Based Variables</td>
<td>28</td>
</tr>
<tr>
<td>Static Variables</td>
<td>28</td>
</tr>
<tr>
<td>Addressing Beyond the 4K Limit</td>
<td>28</td>
</tr>
<tr>
<td>The Pseudo-register Vector (PRV)</td>
<td>29</td>
</tr>
<tr>
<td>Program Control Data</td>
<td>31</td>
</tr>
<tr>
<td>Handling Data Aggregates</td>
<td>31</td>
</tr>
<tr>
<td>Arrays of Structures and Structures of Arrays</td>
<td>32</td>
</tr>
<tr>
<td>Array and Structure Assignments</td>
<td>32</td>
</tr>
<tr>
<td>Handling Flow of Control</td>
<td>32</td>
</tr>
<tr>
<td>Activating and Terminating Blocks</td>
<td>33</td>
</tr>
<tr>
<td>Prolog and Epilog Code</td>
<td>33</td>
</tr>
<tr>
<td>GOTO Statements</td>
<td>39</td>
</tr>
<tr>
<td>Argument and Parameter Lists</td>
<td>43</td>
</tr>
<tr>
<td>Library Calls</td>
<td>43</td>
</tr>
<tr>
<td>DO-Loops</td>
<td>45</td>
</tr>
<tr>
<td>Compiler-Generated Subroutines</td>
<td>46</td>
</tr>
<tr>
<td>Optimization and Its Effects</td>
<td>47</td>
</tr>
<tr>
<td>Examples of Optimized Code</td>
<td>47</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 4. Run-Time Organization</th>
<th>57</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communications Area</td>
<td>57</td>
</tr>
<tr>
<td>Dynamic Storage Allocation</td>
<td>57</td>
</tr>
<tr>
<td>Contents of a Typical Load Module</td>
<td>59</td>
</tr>
<tr>
<td>The Overall Use of Storage</td>
<td>59</td>
</tr>
<tr>
<td>The Multitasking Library</td>
<td>62</td>
</tr>
</tbody>
</table>

| Chapter 5. Run-Time Problem Determination | 63 |
Chapter 6. Debugging Using Dumps

Considerations ..................................... 75
Section 1: How to Obtain a PL/I Dump .......................... 78
CALL PLIDUMP .................................................. 78
Suggested Coding ............................................. 79
Avoiding Recompilation ....................................... 81
Contents of a PL/I Dump ...................................... 82
Section 2: Suggested Debugging Procedures ...................... 87
Debugging Overlaid Storage .................................. 88
Debugging Procedures ........................................ 89
Section 3: Locating Specific Information ........................... 92
Contents .......................................................... 92
Key Areas of a PL/I Dump ..................................... 93
Key Areas of an ABEND Dump ................................. 98
Stand-alone Dumps ............................................. 99
Housekeeping Information in All Dumps ......................... 99
Finding Variables ............................................. 107
Control Blocks and Fields .................................... 108
Section 4: Special Considerations for Multitasking ................. 111
Section 5: Special Considerations for CICS ....................... 111
Section 6: User Exit Considerations ............................. 111
Section 7: SYSTEM Option Considerations ....................... 112

Chapter 7. Using SSF and CSSF Search Arguments ................. 113
Formulating the Search Argument ................................ 113
Using Standardized Keywords .................................. 114

Chapter 8. Submitting an APAR .................................. 115
Materials to Submit ............................................ 115
Original Source Information ................................... 116
Load Libraries Information .................................... 116
Input Data Sets Information .................................. 116
PL/I Compiler and PL/I Library ................................ 116
Compiler Listing .............................................. 117
JCL Listing ..................................................... 117
CMS Terminal Session Log .................................... 117
Linkage Editor Listing ....................................... 118
Run-Time Dump ............................................... 118
Applied Fixes .................................................. 118

Appendix A. Control Blocks ..................................... 119
Area Locator/Descriptor ....................................... 120
Area Descriptor ............................................... 120
Area Variable Control Block .................................. 121
Aggregate Descriptor Descriptor ............................... 122
Aggregate Locator ............................................. 124
Array Descriptor .............................................. 125
CICS Appendage ............................................... 128
Controlled Variable Block ..................................... 130
Data Element Descriptor (DED) ................................ 132
FORMAT DEDs (FEDs) ....................................... 138
Declare Control Block (DLCB) ................................. 140
Dynamic Storage Area (DSA) .................................. 142
Entry Data Control Block ..................................... 146
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environment Block (ENVB)</td>
<td>147</td>
</tr>
<tr>
<td>Event Variable Control Block</td>
<td>151</td>
</tr>
<tr>
<td>File Control Block (FCB)</td>
<td>152</td>
</tr>
<tr>
<td>Fetch Control Block (FECB)</td>
<td>163</td>
</tr>
<tr>
<td>Input/Output Control Block (IOCB)</td>
<td>164</td>
</tr>
<tr>
<td>Label Data Control Block</td>
<td>170</td>
</tr>
<tr>
<td>Library Workspace (LWS)</td>
<td>171</td>
</tr>
<tr>
<td>ON Communications Area (ONCA)</td>
<td>172</td>
</tr>
<tr>
<td>ON Control Block (ONCB)</td>
<td>175</td>
</tr>
<tr>
<td>PLIMAIN</td>
<td>177</td>
</tr>
<tr>
<td>PLISTART Parameter List</td>
<td>178</td>
</tr>
<tr>
<td>Record Descriptor (RD)</td>
<td>180</td>
</tr>
<tr>
<td>String Locator/Descriptor</td>
<td>181</td>
</tr>
<tr>
<td>Structure Descriptor</td>
<td>182</td>
</tr>
<tr>
<td>Symbol Table (SYMTAB)</td>
<td>183</td>
</tr>
<tr>
<td>Task Communication Area (TCA)</td>
<td>190</td>
</tr>
<tr>
<td>TCA Implementation Appendage (TIA)</td>
<td>196</td>
</tr>
<tr>
<td>TCA Tasking Appendage (TTA)</td>
<td>200</td>
</tr>
<tr>
<td>Task Variable (TV)</td>
<td>202</td>
</tr>
</tbody>
</table>

**Appendix B. Record-Oriented Input/Output**

Introduction .......................................................... 203
Summary of Record I/O Implementation .................................. 203
  Access Method ...................................................... 209
File Declaration Statements ........................................... 212
OPEN Statement ....................................................... 213
Transmission Statements (Library-Call I/O) .......................... 219
EVENT Option .................................................................. 224
CLOSE Statements and Implicit Close ................................... 227
Implicit Open for Library-Call I/O ................................... 229
Error Conditions in Transmission Statements .......................... 229
In-Line I/O Statements .................................................. 233

**Appendix C. Stream-Oriented Input/Output**

Introduction .......................................................... 241
File Handling ............................................................ 246
Handling the Conversions ............................................... 249
Handling GET and PUT Statements ...................................... 249
List-Directed GET and PUT Statements .................................. 249
Data-Directed GET and PUT Statements .................................. 254
Edit-Directed GET and PUT Statements .................................. 257
Formatting for Print Files .............................................. 266
Handling Format Options .................................................. 266
Input and Output of Complete Arrays ................................... 266
PL/I Conditions in Stream I/O ........................................... 266
Built-In Functions in Stream I/O ....................................... 268
The COPY Option .......................................................... 268
The STRING Option ....................................................... 269
The Conversational System and Conversation Files ...................... 270
  Conversational Transmitter Modules .................................. 271
  Formatting ............................................................. 271
Summary of Subroutines Used ............................................. 272
  Initializing Modules .................................................. 272
  Director Modules ...................................................... 273
<table>
<thead>
<tr>
<th>Category</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmitter Modules</td>
<td>274</td>
</tr>
<tr>
<td>Formatting Modules</td>
<td>274</td>
</tr>
<tr>
<td>External Conversation Director Modules</td>
<td>275</td>
</tr>
<tr>
<td>Conversational Modules</td>
<td>275</td>
</tr>
<tr>
<td>Miscellaneous Modules</td>
<td>275</td>
</tr>
<tr>
<td><strong>Index</strong></td>
<td><strong>277</strong></td>
</tr>
</tbody>
</table>
Chapter 1. Compiler Overview

The OS PL/I Optimizing Compiler Version 2 analyzes source programs written in the PL/I language and translates these source statements into a series of machine instructions that form an object module. The compiler operates as a problem state under the operating system.

Figure 2. Compiling, Link Editing and Running
Processing a PL/I Program

Figure 2 on page 1 shows the process 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 (that is, 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 the addresses within the code.
4. Running the load module.

The process is not necessarily a continuous one. The program may, for example, be kept in a compiled or link edited form before it is run, and also be run 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 statements 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;
```

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

```
LH 7,88(8,13) Load J into register 7
AH 7,98(8,13) Add K to J
STH 7,96(8,13) Place result in I
```

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

The OS PL/I Optimizing Compiler Version 2 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 resident library routines. These routines may call further library routines from the PL/I library. The following PL/I statements for example, result in a call to a resident library routine.

```
DCL X,Y;
X=SIN(Y);
```
The list below shows the code that would result from such statements:

\begin{align*}
\text{LA} & \quad 14,92(0,13) \quad \text{Place address of Y in register 14} \\
\text{LA} & \quad 15,96(0,13) \quad \text{Place address of X in register 15} \\
\text{STM} & \quad 14,15,88(0,3) \quad \text{Place addresses in argument list} \\
\text{LA} & \quad 1,80,(0,3) \quad \text{Point register 1 at argument list} \\
\text{L} & \quad 15,88(0,3) \quad \text{Load register 15 with the address of a library routine. (This is held in the form of an address constant generated by the compiler and resolved by the linkage editor.)} \\
\text{BALR} & \quad 14,15 \quad \text{Branch to the library routine, which carries out the required function.}
\end{align*}

- Preprocessor

The source program passes to the compiler either directly or through a preprocessor stage (see Figure 2 on page 1). The preprocessor can modify source statements in the program or insert additional source statements in the program before compilation begins. You invoke the preprocessor by specifying the compile-time option MACRO. If the compiler detects an error or the possibility of an error during the preprocessor stage, it prints a message on the pages following the input listing. Thus, there are two sets of messages: one for the preprocessor and one for the compiler. Details of preprocessor and compile-time messages are given in the OS PL/I Version 2 Programming: Messages and Codes.

- Compiler

Under the control of a compile-time option, the compiler optimizes code. Some helpful compile-time options are shown in Figure 3. For more information about these options, see the OS PL/I Version 2 Programming Guide.

<table>
<thead>
<tr>
<th>Needed Documentation</th>
<th>Compile-Time Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Listing</td>
<td>SOURCE</td>
</tr>
<tr>
<td>Cross-reference Listing</td>
<td>XREF</td>
</tr>
<tr>
<td>Attribute Table</td>
<td>ATTRIBUTES</td>
</tr>
<tr>
<td>Aggregate Table</td>
<td>AGGREGATE</td>
</tr>
<tr>
<td>Storage Table</td>
<td>STORAGE</td>
</tr>
<tr>
<td>Compiler Options</td>
<td>OPTIONS</td>
</tr>
</tbody>
</table>

Figure 3 (Part 1 of 2). Compile-Time Options Helpful in Problem Determination
### Link Editing

Link editing links the compiler output with external modules requested by the compiled program. These are PL/I 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.

### Running

The OS PL/I Optimizing Compiler Version 2 produces code that requires a special arrangement of control blocks and registers for the correct run. 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. The PL/I initialization routines handle this.
2. Running the program.
3. Completing the job after the run. This consists of closing any files left open and returning control either to the supervisor or to a calling module. A termination routine handles this.
Chapter 2. Compile-Time Problem Determination

This chapter contains the compile-time problem determination chart. If you are just beginning your problem diagnosis, start by using the chart in Figure 5. Begin with Block 1 and answer the question or perform the action specified, then go to the block indicated by the answer. Some blocks describe an action to be performed and also direct you to the next block.

If you have already made a preliminary diagnosis of your problem and you are familiar with PL/I, you can use the chart index below in Figure 4 to find the information you need.

<table>
<thead>
<tr>
<th>Compile-Time Subject Covered</th>
<th>Block Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abend or Program Check</td>
<td>26</td>
</tr>
<tr>
<td>Search Argument Generation</td>
<td>27</td>
</tr>
<tr>
<td>Message</td>
<td>28</td>
</tr>
<tr>
<td>Search Argument Generation</td>
<td>31</td>
</tr>
<tr>
<td>Loop</td>
<td>32</td>
</tr>
<tr>
<td>Search Argument Generation</td>
<td>33</td>
</tr>
<tr>
<td>Wait</td>
<td>34</td>
</tr>
<tr>
<td>Unusual or Unexpected Output</td>
<td>35</td>
</tr>
<tr>
<td>Search Argument Generation</td>
<td>36</td>
</tr>
<tr>
<td>Performance</td>
<td>37</td>
</tr>
<tr>
<td>Search Argument Generation</td>
<td>38</td>
</tr>
</tbody>
</table>

**Figure 4. Compile-Time Problem Determination Index**

<table>
<thead>
<tr>
<th>Block No.</th>
<th>Question</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Is this a compile-time or a run-time failure?</td>
<td>Compile-time failure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Go to 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Run-time failure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Go to 100 on page 63</td>
</tr>
<tr>
<td>2</td>
<td>Is this a U level message?</td>
<td>Yes 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 3</td>
</tr>
<tr>
<td>3</td>
<td>Is this a problem relating to a message?</td>
<td>Yes 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 4</td>
</tr>
<tr>
<td>4</td>
<td>Is this a loop?</td>
<td>Yes 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 5</td>
</tr>
<tr>
<td>5</td>
<td>Is this a wait?</td>
<td>Yes 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 6</td>
</tr>
<tr>
<td>6</td>
<td>Does the compilation result in some type of unusual or unexpected output?</td>
<td>Yes 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 7</td>
</tr>
</tbody>
</table>

**Figure 5 (Part 1 of 6). Compile-Time Problem Determination Chart**
<table>
<thead>
<tr>
<th>Block No.</th>
<th>Question</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Is this a performance problem?</td>
<td>Yes 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 24</td>
</tr>
<tr>
<td>8</td>
<td>Has the program ever compiled before?</td>
<td>Yes 9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 11</td>
</tr>
<tr>
<td>9</td>
<td>Has anything in the environment been changed?</td>
<td>Yes 12</td>
</tr>
<tr>
<td></td>
<td>(Source changes, release level, maintenance fixes, compile time options,)</td>
<td>No 10</td>
</tr>
<tr>
<td>10</td>
<td>Is the entry from:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>U level message</td>
<td>Yes 26</td>
</tr>
<tr>
<td></td>
<td>Message other than U level</td>
<td>Yes 28</td>
</tr>
<tr>
<td></td>
<td>Loop</td>
<td>Yes 32</td>
</tr>
<tr>
<td></td>
<td>Wait</td>
<td>Yes 34</td>
</tr>
<tr>
<td></td>
<td>Unusual or unexpected output</td>
<td>Yes 35</td>
</tr>
<tr>
<td></td>
<td>Performance</td>
<td>Yes 37</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> If you are here via the “environment changed” route, follow the major symptom code being experienced.</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Make sure PL/I coding rules have been followed. Check and correct any statements causing “E” or “S” level messages. If you are using the “OPTIMIZE(TIME)” or “OPTIMIZE(2)” compile option, recompile using “NOOPTIMIZE.” Is the problem circumvented?</td>
<td>Yes 41</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 10</td>
</tr>
<tr>
<td>12</td>
<td>Has the source code of the program been changed? (This includes compile time options.)</td>
<td>Yes 15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 13</td>
</tr>
<tr>
<td>13</td>
<td>Has any maintenance been applied? (PTFs, fixes)</td>
<td>Yes 17</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 14</td>
</tr>
<tr>
<td>14</td>
<td>Has the release level changed?</td>
<td>Yes 18</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 25</td>
</tr>
<tr>
<td>15</td>
<td>Check and correct any source statements causing “E” or “S” level messages. Be critical of source changes. Is the problem solved?</td>
<td>Yes END</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 16</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> If the problem is solved, but you feel the message was generated in error, follow the “NO” path.</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>If you are using the “OPTIMIZE(TIME)” or “OPTIMIZE(2)” compile option, recompile using “NOOPTIMIZE.” Is the problem circumvented?</td>
<td>Yes 41</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 10</td>
</tr>
<tr>
<td>17</td>
<td>Is(Are) the fix(es) or PTF(s) installed correctly? In other words, were there any system messages while installing and link editing? Search early warning microfiche or INFO/ACCESS, or ask the IBM Support Center Level 1 to search RETAIN for possible PTF errors in the form &quot;PExxxxx&quot;.</td>
<td>Yes 19</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 20</td>
</tr>
<tr>
<td>Block No.</td>
<td>Question</td>
<td>Action</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>18</td>
<td>Is the release installed correctly? (Search early warning microfiche or INFO/ACCESS, or ask the IBM Support Center Level 1 to search RETAIN for any PTFs and errors applicable to this release. Search for PTF errors in the form: “PExxxx”.) Were there any SMP error messages or system messages while installing the release?</td>
<td>Yes 19</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 21</td>
</tr>
<tr>
<td>19</td>
<td>Search early warning microfiche, INFO/ACCESS, or have the IBM Support Center search RETAIN. Information about conducting a search is in Chapter 7, “Using SSF and CSSF Search Arguments” on page 113. Any hits?</td>
<td>Yes 22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 10</td>
</tr>
<tr>
<td>20</td>
<td>Reinstall the PTF or fix and test. Is the problem solved?</td>
<td>Yes END</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 23</td>
</tr>
<tr>
<td>21</td>
<td>Reinstall the release level correctly, plus any PTFs or fixes that are applicable, and test. Is the problem solved?</td>
<td>Yes END</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 23</td>
</tr>
<tr>
<td>22</td>
<td>Apply applicable fix(es) from RETAIN and test. Is the problem solved?</td>
<td>Yes END</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 23</td>
</tr>
<tr>
<td>23</td>
<td>Have the symptoms changed?</td>
<td>Yes 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 10</td>
</tr>
<tr>
<td>24</td>
<td>Something has apparently been overlooked. A failure must have occurred, and it must have been one of the previously mentioned items. Review the compiler output again. If the problem does not fit any of the stated symptoms, go to Block 41.</td>
<td>41</td>
</tr>
<tr>
<td>25</td>
<td>Obviously something has been changed. Investigate the system control program for possible changes such as fixes, PTFs, library updates or reorganization, etc.</td>
<td>27</td>
</tr>
<tr>
<td>26</td>
<td>Compiler program checks will produce one of the following messages:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IEL0001I U PREPROCESSOR ERROR n DURING PHASE p</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IEL0230I U COMPILER ERROR NUMBER n DURING PHASE p</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IEL0970I U COMPILER CANNOT PROCEED. ERROR n DURING PHASE p. CORRECT SOURCE AND RECOMPILDE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Note: Details of compiler error numbers and sometimes recommended programmer actions are in your OS PL/I Version 2 Programming: Messages and Codes manual. Go to Block 27.</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5 (Part 3 of 6). Compile-Time Problem Determination Chart
<table>
<thead>
<tr>
<th>Block No.</th>
<th>Question</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>Search early warning microfiche, INFO/ACCESS, or have the IBM Support Center search RETAIN using the component-id 5668909, the error number from the message and phase-id from message. Information about conducting a search is in Chapter 7, “Using SSF and CSSF Search Arguments” on page 113. Any hits?</td>
<td>Yes 39  No 41</td>
</tr>
<tr>
<td>28</td>
<td>Is this an “E” or “S” level diagnostic message?</td>
<td>Yes 29  No 30</td>
</tr>
<tr>
<td>29</td>
<td>Look up the message in OS PL/I Version 2 Programming: Messages and Codes. These messages indicate the compiler has detected error condition(s) in the source statements. Compilation may be complete, but the object program may not run correctly. Check and correct any statements in error. Rerun the program. Does the message still occur?</td>
<td>Yes 30  No END</td>
</tr>
<tr>
<td>30</td>
<td>If you are using the “OPTIMIZE(TIME)” or “OPTIMIZE(2)” compile option, recompile with “NOOPTIMIZE”. Is the problem circumvented?</td>
<td>Yes 41  No 31</td>
</tr>
<tr>
<td>31</td>
<td>Search early warning microfiche, INFO/ACCESS or have the IBM Support Center search RETAIN using component-id 5668909 and MSGIELxxxxl. Information about doing your search is in Chapter 7, “Using SSF and CSSF Search Arguments” on page 113. Any hits?</td>
<td>Yes 39  No 41</td>
</tr>
</tbody>
</table>

Figure 5 (Part 4 of 6). Compile-Time Problem Determination Chart
<table>
<thead>
<tr>
<th>Block No.</th>
<th>Question</th>
<th>Action</th>
</tr>
</thead>
</table>
| 32       | If a loop appears to be occurring, use a system trace facility or instruction step mode to capture all, or at least part, of the loop addresses. Then cancel the job with a dump. Find the current phase in the dump as follows: • Register 13 points to the communications area (XCOMM). • To check, look at the field at offset X'90' from register 13. This field should contain the first source input record. If register 13 has been corrupted, search for this field to locate XCOMM. • The current phase name is at offset X'4AB' from the beginning of XCOMM. This field contains two letters. Adding "IELO" before these two letters and will give you the name of the current phase. • The phase start address is at offset X'434' from XCOMM. If you are using the "OPTIMIZE" compile option, recompile using "NOOPTIMIZE." Phase "IELOIE" may appear to be in a loop if a large number of "BYNAME" assignments are used or if the source does an assign to a large "PICTURE" statement. Phase "IELOIK" may appear to be in a loop sorting data names for the XREF table. Give the compiler a little more time. Other loops may be caused by: • Using a colon instead of a semicolon • Not enough storage • Not enough time. Is the problem circumvented? | Yes 41  
No 33 |
| 33       | Search early warning microfiche, INFO/ACCESS, or have the IBM Support Center search RETAIN using: • component-id 5668909 • LOOP and module name(s) in which loop occurs. Information about doing your search is in Chapter 7, “Using SSF and CSSF Search Arguments” on page 113. Any hits? | Yes 39  
No 41 |
| 34       | The only waits the compiler issues are for I/O. Wait states should be investigated from the system control viewpoint: • Check to see that the region running PL/I is not waiting for a resource owned by another region. • See if there are any system messages. If you still suspect the PL/I compiler is causing the wait, go to Block 41. | 41 |

Figure 5 (Part 5 of 6). Compile-Time Problem Determination Chart
<table>
<thead>
<tr>
<th>Block No.</th>
<th>Question</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>Be sure all appropriate compile options are specified.</td>
<td>Yes 41</td>
</tr>
<tr>
<td></td>
<td>If you are using the “OPTIMIZE(TIME)” or “OPTIMIZE(2)” compile option, recompile with</td>
<td>No 36</td>
</tr>
<tr>
<td></td>
<td>“NOOPTIMIZE”.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Is the problem circumvented?</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>Search early warning microfiche, INFO/ACCESS, or have the IBM Support Center search RETAIN</td>
<td>Yes 39</td>
</tr>
<tr>
<td></td>
<td>using component-id 5668909 and INCORROUT, a word describing what output is incorrect.</td>
<td>No 41</td>
</tr>
<tr>
<td></td>
<td>Information about doing your search is in Chapter 7, “Using SSF and CSSF Search Arguments”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>on page 113.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Any hits?</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>Performance problems usually show up after some environment change; if not a maintenance</td>
<td>Yes 41</td>
</tr>
<tr>
<td></td>
<td>change, then perhaps a source code or compile option change.</td>
<td>No 38</td>
</tr>
<tr>
<td></td>
<td>Review these items.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>If you are using the “OPTIMIZE(TIME)” or “OPTIMIZE(2)” compile option, recompile with</td>
<td></td>
</tr>
<tr>
<td></td>
<td>“NOOPTIMIZE”.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Is the problem circumvented?</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>Search early warning microfiche, INFO/ACCESS, or have the IBM Support Center search RETAIN</td>
<td>Yes 39</td>
</tr>
<tr>
<td></td>
<td>using component-id 5668909.</td>
<td>No 41</td>
</tr>
<tr>
<td></td>
<td>Information about doing your search is in Chapter 7, “Using SSF and CSSF Search Arguments”</td>
<td></td>
</tr>
<tr>
<td></td>
<td>on page 113.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Any hits?</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>Apply the fix(es), circumvention from RETAIN, and test.</td>
<td>Yes END</td>
</tr>
<tr>
<td></td>
<td>Is the problem solved?</td>
<td>No 40</td>
</tr>
<tr>
<td>40</td>
<td>Did the symptoms change?</td>
<td>Yes 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 41</td>
</tr>
<tr>
<td>41</td>
<td>Contact the IBM Support Center for assistance.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Have available the following documentation:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Compilation listing with LIST, SOURCE, XREF, STMT, ESD, and MAP options specified</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- The TSO or CMS command sequence or the JCL used to run the job</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Dump (if applicable)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Preprocessor input (if applicable)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- List of applied fixes</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5 (Part 6 of 6). Compile-Time Problem Determination Chart
Chapter 3. Compiler Output

This chapter describes the part of the load module generated by the compiler. The compiler output is a relocatable object module consisting of a series of 80-column records. These records contain either machine instructions, constants, or external or internal addresses that the linkage editor resolves. The records are:

**TXT records**
- Contain machine instructions or constants.

**RLD records**
- Contain internal addresses that require updating for a load module.

**ESD records**
- Contain external names to be resolved (bound) with other programs and data areas.

The compiler produces two main control sections:
- The program control section, holding the executable instructions translated from the PL/I program.
- The static internal control section holding constants, addresses, and static variables.

The compiler also generates a number of other control sections. These handle certain housekeeping functions or they are used for external data. This external data may have identical control sections generated for it by other compilations.

Storage for workspace and automatic variables is acquired during execution, normally by the prolog code that is executed at the start of every block.

The output from the compiler is shown in Figure 6 on page 13, 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.
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 sub-routine used.

3. Dummy sections
Pseudo-register vector
A dummy section used to address files, controlled variables and FETCHable entry constants.

Program initialization uses the two control sections, PLISTART and PLIMAIN. PLISTART holds the address of the library initialization routine IBMBPIR, which is 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 ends. 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 run together, control passes to the first procedure processed by the linkage editor.

Appendix A, “Control Blocks” on page 119 gives the format of PLIMAIN and PLISTART.

If you use the compile-time FLOW option, 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.
Figure 6. The Output from the Compiler
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. A short discussion of the effects of optimization completes the chapter.

Listing Conventions

Figure 7 shows all the program listing information that the compiler can produce. It also shows the relevant compile-time options and summarizes the information produced if these options are specified.

This chapter describes the contents of the static-storage map and the object-program listing. OS PL/I Version 2 Programming Guide gives information on the other items generated.

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

Figure 7. Contents of Listing and Associated Compile-Time Options

Note to Figure 7:

1. External references within library modules are not included.
Static-Storage Map

The static-storage map is a formatted listing of the contents of the static internal and static external control sections. You obtain this listing by specifying the MAP option in the PROCESS statement. 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 8 on page 17.

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 descriptor
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
SYMBOL TABLE ELEMENT Address of symbol table
SYMBOL TABLE...xxx Symbol table for xxx
SYMTAB DED...xxx Symbol table DED for xxx
USER LABEL xxx
xxx

Source program label for 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.)
SOURCE

EXAMPLE: PROC OPTIONS(MAIN) REORDER;
DCL X(10),Y,Z INITIAL (0);
GET EDIT(X,Y)(F(3),X(11));
DO I = 1 TO Y;
    Z = Z*X(I);
END;
PUT EDIT(Z)(A);
END;

STATIC INTERNAL STORAGE MAP

000000 E0000100 PROGRAM ADCON 000000 0000000000000000 DCLCB
000004 00000908 PROGRAM ADCON 000000 0000000000000000
000008 0000099A PROGRAM ADCON 000000 0000140005E2E8
00000C 000090A4 PROGRAM ADCON E2C90560
000010 000090A4 PROGRAM ADCON
000014 000090A4 A..IELCIX
000018 000090A4 A..IELCIX 000000 FFFFFFFC41201000 DCLCB
00001C 000090A4 A..IBMBCACA
000020 000090A4 A..IBMBCEDB 02D7000800000000
000024 000090A4 A..IBMBCFD 00000140000E2E8
000028 000090A4 E..IBMBCFH
00002C 000090A4 A..IBMBCVY
000030 000090A4 A..IBMBOCLA
000034 000090A4 A..IBMBOCLC
000038 000090A4 A..IBMBSAOA
00003C 000090A4 A..IBMBSEDB
000040 000090A4 A..IBMBS1IA
000044 000090A4 A..IBMBS1IT
000048 000090A4 A..IBMBS1IA
00004C 000090A4 A..IBMBS1IA
000050 000090A4 A..IBMBS1OA
000054 000090A4 A..IBMBS1OT
000058 000090A4 A..IBMBSXCA
00005C 000090A4 A..STATIC
000060 000090A4 0000000800000000 DED..X
000064 000090A4 0000000800000000 FED
000068 000090A4 0000000000000000 FED
00006E 000090A4 0000000000000000 FED
000072 000090A4 0000000000000001 CONSTANT
000074 000090A4 0000000000000001 CONSTANT
000076 000090A4 0000000000000001 CONSTANT
000078 000090A4 0000000000000001 CONSTANT
00007C 000090A4 0000000000000001 CONSTANT
000080 000090A4 0000000000000001 CONSTANT
000084 000090A4 0000000000000001 CONSTANT
000088 000090A4 0000000000000001 CONSTANT
00008C 000090A4 0000000000000001 CONSTANT
000090 000090A4 0000000000000001 A..DCLCB
000094 000090A4 0000000000000001 A..DCLCB
000098 000090A4 0000000000000001 A..DCLCB
00009C 000090A4 0000000000000001 A..DCLCB
0000A0 000090A4 0000000000000001 A..DCLCB
0000A4 000090A4 0000000000000001 A..DCLCB
0000A8 000090A4 0000000000000001 COMPILER LABEL CL.11

Figure 8. Example of Static Storage Listing
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, a translation of these instructions into a form that resembles assembler language, and number of comments, such as the statement number. The format of this listing is shown in Figure 9 on page 20.

The format has blocks of code headed by the number of the statement in the PL/I program to which they are equivalent. When optimization results 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. As a result, the order of statements frequently differs 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. Another constant follows this which contains the length of the procedure name. Next, the name of the procedure appears as a comment, followed by code under the heading "REAL ENTRY." At this point, the code is entered. The second section of code is the prolog, which carries out various housekeeping tasks and is described more fully later in this chapter. The message "PROCEDURE BASE." marks the end of the prolog. Following this is 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—heads the initialization code for an entry point to a procedure.
- PROLOG BASE—identifies the start of the prolog 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 the address 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 —identifies the end of a procedure.
• BEGIN BLOCK xxx—indicates the start of the begin block with label xxx.
• BEGIN BLOCK NUMBER xx—indicates the start of the begin block with number xx.
• END BLOCK —indicates the end of the begin block.
• 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
• END INTERLANGUAGE PROCEDURE xxx—identifies the end of encompassing procedure xxx.
• 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 NUMBER xx—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 was moved from within a loop by the optimization process.
• CODEMOVED_FROM STATEMENTNUMBER 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.
• METHOD OR ORDER OF CALCULATING EXPRESSIONS CHANGED—indicates that the order of the code following was changed to optimize the object code.
PL/I OPTIMIZING COMPILER EXAMPLE: PROC OPTIOTIS(TAL) REORDER;

/* REAL ENTRY */

* STATEMENT NUMBER 1

/* END OF COMPILER GENERATED SUBROUTINE */

Figure 9 (Part 1 of 2). Part of an Object Program Listing
**Figure 9 (Part 2 of 2). Part of an Object Program Listing**

In certain cases, the compiler uses mnemonics to identify the type of operand in an instruction, and, where applicable, follows the mnemonic 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** Identifier created for an otherwise unlabeled block
- **CL.nn** Compiler-generated label
- **D.** Descriptor
- **DED.** Data element descriptor
- **HOOK..ENTRY** PL/TEST block entry hook
- **HOOK..BLOCK-EXIT** PL/TEST block exit hook
- **HOOK..PGM-EXIT** PL/TEST program exit hook
- **HOOK..PRE-CALL** PL/TEST pre-call or function reference hook
- **HOOK..INFO** PL/TEST Additional pre-call hook information
- **HOOK..POST-CALL** PL/TEST post call or function reference hook
- **HOOK..STMT** PL/TEST statement hook
- **HOOK..IF-TRUE** PL/TEST IF true hook
- **HOOK..ELSE** PL/TEST ELSE hook
- **HOOK..WHEN** PL/TEST WHEN true hook
- **HOOK..OTHERWISE** PL/TEST OTHERWISE true hook
- **HOOK..LABEL** PL/TEST label hook
- **HOOK..DO** PL/TEST iterative DO hook
- **HOOK..ALLOC** PL/TEST ALLOCATE controlled hook
- **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 8 on page 17.

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

1. Addresses in static CSECT and code CSECT
2. Addresses of library modules
3. Addresses of entry points
4. Addresses of label constants that may be assigned to label variables
5. 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):

<table>
<thead>
<tr>
<th>Constants</th>
<th>Constant Values Used by Compiled Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ONCBs</td>
<td>Control blocks used in error handling.</td>
</tr>
<tr>
<td>Descriptors, locators and DEDs</td>
<td>Control information used by compiled code and library.</td>
</tr>
<tr>
<td>(data element descriptors)</td>
<td></td>
</tr>
<tr>
<td>Symbol table address vector</td>
<td>Control information used in data-directed I/O.</td>
</tr>
</tbody>
</table>

Figure 10. Constants Pool Contents

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

The next section of the static internal control section holds the static variables. These are held in size order, with the smallest being first.

The final section of the static internal control section contains branch tables for those select-groups for which optimized code has been produced, the symbol table vectors and symbol table for PLTEST, the truncated statement number tables containing GOSTMT and GONUMBER data, and the TIMESTAMP data (if this option has been specified at installation time).
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 are common to all source programs.

This 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 11.

<table>
<thead>
<tr>
<th>Register Number</th>
<th>Dedicated Registers</th>
<th>Work Registers (plus special use)</th>
<th>Preferred Registers</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>General</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>General + address of parameter list</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Address of program base</td>
<td></td>
<td>Saved during in-line record I/O and TRT instructions</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Address of static base</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>General</td>
<td></td>
<td>Address of temporary base if DSA size greater than 3896 bytes</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>General + static back-chain on entry to procedure</td>
<td>Preferred register for DO-loop control variable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>General</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>General</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>General</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>General</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>General</td>
<td>Preferred register for DO-loop control when BXLE instruction is used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>General</td>
<td>Preferred register for DO-loop control when BXLE instruction is used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Address of TCA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Address of current DSA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>General + branch-and-link to library and other routines</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>General + branch-and-link to library and other routines</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 11. Register Usage in Compiled Code
The compilers use four general registers as bases for addressing various types of data; these registers are *dedicated registers*. The rest of the registers are used as required. These are *work registers*.

The dedicated registers are:

- **R2** Program base
- **R3** Static base
- **R12** TCA pointer
- **R13** DSA pointer

By arranging the dedicated registers this way, the compiled code uses five even/odd work register pairs. These registers are (0,1), (6,7), (8,9), (10,11), and (14,15).

The compiler always uses certain registers for special tasks. The compiler preferentially uses other registers for other tasks when they are available. These tasks are shown in Figure 11 on page 24.

**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. Register 2 is not used 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. During these times, 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 compiler lists the items found in this control section in any particular program in the static-storage map. (See "Static Internal Control Section" on page 22). When the static control section is larger than 4K bytes, an additional base register is used.

**Register 12—TCA**: Offsets from register 12 are used to address the various fields in the TCA. Its format is shown in Appendix A, "Control Blocks" on page 119.

**Register 13—Current DSA**: Register 13 points to the current DSA. Register 13 addresses 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 refer to the housekeeping fields held in every DSA. Appendix A, "Control Blocks" on page 119 shows the format of the housekeeping information in a DSA.

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

**Work Registers**

Figure 11 on page 24 shows special or preferred uses for work registers. Registers with special uses are free and always used for the special uses. Registers with preferred uses are 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 register usage in compiled code. Figure 12 shows library register usage.

In both library and compiled code usage, register 12 points at the TCA, and register 13 at the current DSA. Both library subroutines and compiled code use registers 14 and 15 to branch and link between routines.

**Note:** Registers 14 through 4 are normally saved by the library because most library subroutines use only these registers. You can save time by reducing save-restore requirements. However, some library routines also save one or more of registers 5 through 11.

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Work register</td>
</tr>
<tr>
<td>2</td>
<td>Work register</td>
</tr>
<tr>
<td>3</td>
<td>Program base register (dedicated)</td>
</tr>
<tr>
<td>4</td>
<td>Work register</td>
</tr>
<tr>
<td>5</td>
<td>Work register</td>
</tr>
<tr>
<td>6</td>
<td>Work register</td>
</tr>
<tr>
<td>7</td>
<td>Work register</td>
</tr>
<tr>
<td>8</td>
<td>Work register</td>
</tr>
<tr>
<td>9</td>
<td>Work register</td>
</tr>
<tr>
<td>10</td>
<td>Work register</td>
</tr>
<tr>
<td>11</td>
<td>Work register</td>
</tr>
<tr>
<td>12</td>
<td>TCA pointer (dedicated in both library and compiled code)</td>
</tr>
<tr>
<td>13</td>
<td>DSA pointer</td>
</tr>
<tr>
<td>14</td>
<td>Work register (always used for branch-and-link to other routines)</td>
</tr>
<tr>
<td>15</td>
<td>Work register (used with register 14 for branch-and-link)</td>
</tr>
</tbody>
</table>

Figure 12. Library Register Usage
Handling and Addressing Variables and Temporaries

Automatic Variables

The compiler allocates storage for automatic variables on a procedure or begin-block basis. If the length of the variable is known during compilation then the compiler allocates storage within the DSA of the block in which they are declared. However, if the length of the variable is not known until execution, then the compiler allocates storage in variable data areas (VDAs). VDAs are held in the last-in/first-out storage stack and are acquired in the prolog code after the DSA has been acquired. The DSA is acquired in the same way and is described in “Prolog” on page 33.

If automatic variables are used in the block in which they are declared and are held in the DSA, the compiler addresses them from register 13. If they are held in a VDA, the compiler addresses them from a separate base set up for the VDA.

Automatic variables known in any block are those that are declared in that block, or in any encompassing blocks. The method used to address automatic variables in outer blocks is a static back-chain.

The compiler-generated prolog for a procedure saves the address of the static back-chain DSA. This address can then be accessed from register 13. Frequently, the value is retained in the register and not reloaded when the variable is accessed. Typical code is:

```
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 Pseudo-register Vector (PRV)” on page 29. When no allocations of the controlled variable are made, the PRV offset points to the dummy File Control Block (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, “Control Blocks” on page 119. Storage for controlled variables is allocated in separate heap 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. Storage for based variables which appear in an ALLOCATE statement is allocated in HEAP or in a specified AREA.

**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(8,13) Place address of 4K boundary in register 6.
AH 7,96(8,6) 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, fetched procedures, and files, PL/I uses a control block called the pseudo-register vector (PRV). 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, fetched procedures, or files are placed in the PRV.

For an introduction to pseudo-registers, see OS/VS Linkage Editor and Loader, or MVS/Extended Architecture Linkage Editor and Loader.

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 when the file is opened during execution. The use of the linkage editor means that FETCHed procedures cannot use controlled variables or files, except SYSPRINT.

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

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 procedures so that their names are unique. Figure 13 on page 30 summarizes the use of the PRV.
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 13. Use of the Pseudo-register Vector (PRV)

The Location of the PRV

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

Under Multitasking: 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, 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 was 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 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 X'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, "Control Blocks" on page 119.

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 there had been such an element 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.

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.
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;
are held in the order

\[
\begin{array}{cccc}
\end{array}
\]

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);
are held in the order

\[
\begin{array}{cccc}
S.B(1) & S.B(2) & S.C(1) & S.C(2) \\
\end{array}
\]

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.

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 is made to it as a whole, and the elements are not moved one at a time. Similarly, structures that are contiguous and have the same format are moved as a whole.

Handling Flow of Control

In PL/I, five types of statement can result in nonconsecutive 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.

Activating and Terminating Blocks

BEGIN, CALL, END, and RETURN statements, and function references all result in the activation or termination of blocks. The block structure of PL/I 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 prolog 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 prolog 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 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.

Prolog and Epilog Code

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

Prolog

The prolog appears on the object-program listing between REAL ENTRY and PROCEDURE BASE or BLOCK BASE. Every prolog 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 prolog 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.

An example of prolog code is shown in Figure 14 on page 35. More information about constants is in the "Communicating with Assembler Language Programs" chapter in the *OS PL/I Version 2 Programming Guide*. 
Figure 14. Typical Prolog Code

After saving the registers, the prolog 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 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 back-chains are set up. The dynamic back-chain, which points to the DSA of the calling or preceding block, and the static back-chain, which points to the DSA of the statically encompassing block. For the main procedure, the dynamic back-chain points to the dummy DSA, and the static back-chain is set to zero. The address of the statically encompassing block is passed in register 5.

Static back-chains 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 prolog is considerably different from the one described above.

The format of the DSA is shown in Figure 15 on page 37; full details are shown in Appendix A, "Control Blocks" on page 119.
<table>
<thead>
<tr>
<th>Housekeeping information</th>
</tr>
</thead>
<tbody>
<tr>
<td>See Appendix A, &quot;Control Blocks&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Items &lt; 9 bytes in length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Held in alignment order:</td>
</tr>
<tr>
<td>doubleword</td>
</tr>
<tr>
<td>fullword</td>
</tr>
<tr>
<td>halfword</td>
</tr>
<tr>
<td>byte</td>
</tr>
<tr>
<td>bit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Items 9 - 2048 bytes in length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Held in alignment order as above</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Items &gt; 2048 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Held in alignment order as above</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter storage area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addresses of any parameters passed to the associated procedure are stored here</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register bind storage area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used by compiled code when registers must be saved</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Local temporary storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used for temporaries required for duration of statement</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Global temporary storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used by temporaries required for duration of block</td>
</tr>
</tbody>
</table>

Figure 15. Contents of Typical Compiled Code DSA
Epilog

Epilog 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 epilog code is shown in Figure 16.

Epilog code for main procedure

| LR 0,13 | Save current DSA address |
| L 13,4(0,13) | Back-chain |
| 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 |

Epilog code for subroutine or begin block

| L 13,4(0,13) | Back-chain |
| LM 14,12,12(13) | Restore registers of preceding block |
| BR 14 | Return |

Figure 16. Epilog Code

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). Epilog code for a main procedure therefore takes a different form to that generated for a subroutine.

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.

This example is for a call to an external procedure without parameters.

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00005E</td>
<td>IB 11</td>
<td>No parameters, so clear Reg1</td>
</tr>
<tr>
<td>000060</td>
<td>IB 55</td>
<td>External proc, so no static back-chain</td>
</tr>
<tr>
<td>000062</td>
<td>F0 3 024</td>
<td>Pick up address of procedure</td>
</tr>
<tr>
<td>000066</td>
<td>EF</td>
<td>Branch to procedure</td>
</tr>
<tr>
<td></td>
<td>BALR 14,15</td>
<td>Branch to procedure</td>
</tr>
</tbody>
</table>
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. In those cases where the compiler builds the parameter list in internal static, the typical code might be:

```
0001FE 41 90 6 0B4 LA 9,B
000202 59 90 3 0BC ST 9,188(0,3)
000206 41 90 6 0B8 LA 9,A
00020A 59 90 3 0C9 ST 9,192(0,3) Set up parameter list
00020E 18 56 90 188(0,3) Load static back-chain 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
```

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 back-chain must be made to correct DSA. A back-chain is made through any BEGIN blocks. The depth of nesting can be determined during compilation, so the back-chain can be loaded the required number of times before the branch is made.

Typical code would be:

```
0003F0 58 D0 004 L 13,4(0,13) Pick up DSA back-chain
0003F4 98 EC D 00C LM 14,12,12(13) Restore registers
0003FB 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.

GOTO Statements

Depending on whether the GOTO statement branches to a label within the block or external to the block, then the branching has different implications. 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.

This interpretive code is held in the TCA. The compiled code branches to the interpretive code to implement a GOTO that may transfer control out of the block. This TCA code determines 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 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.

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 is:

```
000F1A 47 F0 2 OC8 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 other 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 normally 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). Through registers 14 and 15 the GOTO code receives either the contents of the label variable or the equivalent information for a label constant. This equivalent information is the address where 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 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 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 left the block acquired a VDA. The start-of-block NAB value is retained in the DSA and is
known as the end-of-prolog 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-prolog 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:

\[
\text{GOTO label-constant (out of block)}
\]

\[
\begin{align*}
000026 & \ 18 \ E6 \ \text{LR} \ 15,6 \ \text{Place address of DSA in R15} \\
000028 & \ 41 \ E0 \ 3 \ 088 \ \text{LA} \ 14,136(0,3) \ \text{Place address of label constant in R14} \\
00002C & \ 47 \ F0 \ C \ 080 \ B \ 128(0,12) \ \text{Branch to GOTO code in TCA}
\end{align*}
\]

**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:

\[
\text{GOTO label-variable}
\]

\[
\begin{align*}
0000D0 & \ 98 \ EF \ D \ 0A8 \ \text{LM} \ 14,15,168(13) \ \text{Load R14 and R15 with label variable} \\
0000D4 & \ 47 \ F0 \ 0 \ 080 \ B \ 128(0,12) \ \text{Branch to GOTO code in TCA}
\end{align*}
\]

**Errors When Using Label Variables**

Although it is invalid PL/I, it is possible for a GOTO statement using a label vari-
able to result in transfer of control to an inactive block. The optimizing compi-
ler has no method of checking such errors, and the consequences are
unexpected. 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 verifies this address to be the address of an active DSA, and acts accordingly. Three possibilities arise:

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

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

3. The original DSA has been overwritten with other information. Again, the results are not predictable. When PL/I determines that the data in the DSA is not another DSA, ERROR condition code 9002 is raised.

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**

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 prolog 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.

If CHECK enablement has to be changed because of a GOTO, the interpretive GOTO routine calls the library routine IBMBPGO to reset check enablement.
Argument and Parameter Lists

In PL/I, a parameter list is a list of the items a program expects to receive; 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.

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.

Arguments are passed in an argument list addressed by register 1. For nonreentrant, nonrecursive code, the list is set up in static storage and completed by the compiler if the values are are known at compile time. If the procedure is reentrant, recursive, or fetched, the list is moved into the temporary storage area in the DSA before the call is made; otherwise the parameter list is moved into automatic storage.

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 14 on page 35) 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 occur in object program. All library calls that appear in the object listing are to resident PL/I library modules. Transient PL/I library routines are called by routines in the resident library routines.

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.

Figure 17 on page 44 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 “IBM,” followed by five 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.
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.

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

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

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

Figure 17. Examples of Library Calling Sequences

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 17, 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.

Addressing the Subroutines

As can be seen in example 1 of Figure 17, 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 Figure 17. 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 18 on page 45).
<table>
<thead>
<tr>
<th>Offset from Start of TCA (Register 12)</th>
<th>Offset from Start of TCA (Register 12)</th>
<th>Name of Module Entry Point</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>72 48</td>
<td>IBMBPGRD</td>
<td>Stack overflow routine to get VDA</td>
<td></td>
</tr>
<tr>
<td>84 54</td>
<td>IBMBEFL</td>
<td>FLOW module</td>
<td></td>
</tr>
<tr>
<td>108 6C</td>
<td>IBMBPGRA</td>
<td>Get non-LIFO dynamic storage</td>
<td></td>
</tr>
<tr>
<td>112 70</td>
<td>IBMBPGRB</td>
<td>Free non-LIFO dynamic storage</td>
<td></td>
</tr>
<tr>
<td>116 74</td>
<td>IBMBPGRC</td>
<td>Stack overflow routine for prolog</td>
<td></td>
</tr>
<tr>
<td>120 78</td>
<td>IBMBERRB</td>
<td>Error handler software interrupt</td>
<td></td>
</tr>
<tr>
<td>264 108</td>
<td>IBMBJWTA</td>
<td>WAIT module</td>
<td></td>
</tr>
<tr>
<td>268 10C</td>
<td>IBMBTOCA</td>
<td>Completion pseudo-variable routine</td>
<td></td>
</tr>
<tr>
<td>272 110</td>
<td>IBMBTOCB</td>
<td>Event variable assignment routine</td>
<td></td>
</tr>
</tbody>
</table>

Figure 18. Offsets Where Addresses of Library Modules Are Held in the TCA

**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*

```plaintext
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 5,1  Place 1 in I
   CL.1  EQU  *
  ...
   ...
   ...
   ...
   LH  5,1
   AH 5,596(0,3)  Increment and
   STH 5,1  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
IELCGOG Stream I/O output—provides address of target of next edit-directed data or format item
IELCGOH 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 '0' bits
IELCGBO  Test for '1' 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 produces 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.

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 are expansions of the examples shown in the OS PL/I Version 2 Programming Guide. The examples do not 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 \quad A=B+C;
3 \quad \text{IF } X<Y \text{ THEN } X=Y;
4 \quad D=B+C;

* STATEMENT NUMBER 2
00005E 78 00 D 0BC
000062 7A 00 D 0C0
000066 70 00 D 0BB

* STATEMENT NUMBER 3
00006A 78 60 03C4
00006E 79 60 03C8
000072 47 80 2 020
000076 78 60 03C8
00007A 70 60 03C4

* STATEMENT NUMBER 4
00007E 00 00 00BC

* CALCULATION OF COMMONED EXPRESSION FOLLOWS
00006E 70 00 00BC

Example 2: Value held in temporary storage: Source program

2 \quad A=B+C;
3 \quad \text{IF } X<Y \text{ THEN } A=6;
4 \quad D=B+C;

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

Object program

* STATEMENT NUMBER 2
00005E 78 00 D 0BC
000062 7A 00 D 0C0
000066 38 29
000066 70 29 D 0BB

* STATEMENT NUMBER 3
00006C 78 60 03C4
000070 79 60 03C8
000074 47 80 2 022
000078 78 29 3 01C
00007C 70 29 0 DB8

* STATEMENT NUMBER 4
000089 00 00 00BC

* CALCULATION OF COMMONED EXPRESSION FOLLOWS
000069 70 00 00BC
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
00005E  48 E0 D 0BA  LH 14,N
000062  18 BE       LR 11,14
000064  48 A0 3 018  LH 10,24(0,3)
000066  18 5A       LR 5,10
00006A  40 50 D 088  STH 5,I
00006E  19 5B       CR 5,11
000070  47 20 2 026  BH CL.3
000074  CL.2       CL.2 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

* STATEMENT NUMBER 2
000074  48 60 3 01A  LH 6,26(0,3)
000078  40 60 D 0BC  STH 6,J
Elimination of Unreachable Statements

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;
       ELSE C=B*C;
7      ELSE C=B*C;
8   LABEL:  X=X+1;
Object program

* STATEMENT NUMBER 5
000008A 47 F8 2 028 B LABEL

* STATEMENT NUMBER 8

* STATEMENT LABEL LABEL
000008E 78 60 D 0AC LE 6,X
000092 7A 60 3 018 AE 6,24
000096 78 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
0000062 78 20 D 0A4 LE 2,B
0000066 3A 22 AER 2,2
000006A 7A 20 D 0A4 AE 2,B
000006E 78 20 D 0A8 STE 2,X

or

6 X=3*B**2

Object program

* STATEMENT NUMBER 6
00000E2 78 40 D 0BC LE 4,B Load B
00000E6 3C 44 MER 4,4 B**2
00000EB 38 64 LER 6,4
00000EA 3A 66 AER 6,6 2*B**2
00000EC 3A 64 AER 6,4 3*B**2
00000EE 78 60 D 0B8 STE 6,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 19 on page 53, 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 incre­
mented. 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 instruc­
tion. 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.
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
00005E 48 60 3 018 LH 6,24(0,3) Pick up 1 from static
000062 40 60 088 STH 6,1 Place in I

* INITIALIZATION CODE FOR OPTIMIZED LOOP FOLLOWS

* CODE MOVED FROM STATEMENT NUMBER 5
000066 48 E0 3 01A LH 14,26(0,3) Load 4 into R14 from static
000066 48 E0 3 01C LH 8,28(0,3) Load 40 into R8 from static
000068 18 B8 LR 11,8 Load 40 into R11 for BXLE
000070 18 AE LR 10,14 Load 4 into R10
000072 18 5E LR 5,14 Load 4 into R5

* CONTINUATION OF STATEMENT NUMBER 4
000074 CL.2 EQU *

* STATEMENT NUMBER 5
000074 78 05 0 08C LE 0,VO..B(5) Pick up VO..B+R5
000078 70 05 0 0E4 STE 0,VO..C(5) Place in VO..C+R5

* STATEMENT NUMBER 6
00007C 87 5A 2 016 BXLE 5,10,CL.2 Increment R5 by 4, test
for end of loop, and
branch or continue

Figure 19. Modification of DO-Loop Control Variable

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 20 on page 54, 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.
Source program

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

Object program

* STATEMENT NUMBER 2
000062 78 80 D 0A0 LE 0,A Pick up A
000066 79 80 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 080 LE 6,Y
00007E 7A 60 D 084 AE 6,Z X=Y+Z
000082 70 60 6 0AC STE 6,X
000086 CL.2 EQU *

Figure 20. Branching Around Redundant Expressions

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 of 768 will be picked up from the same address in both cases. Similarly, compiler-generated control descriptors 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 21 on page 55, constant '123' is stored in a different form for assignment, multiplication, and exponentiation.
Source program

2
3
4
5

X=123;
Y=123*Z;
V=V**123;
A=123;

/*COMMONED ITEM*/

Object program

00005E 78 00 3 01C LE 0,28(0,3) /*COMMONED ITEM*/
000062 70 00 D 088 STE 0,X

* STATEMENT NUMBER 3

000066 78 20 D 0C9 LE 2,Z
00006A 7C 20 3 01C ME 2,28(0,3)
00006E 70 20 D 0BC STE 2,Y

* STATEMENT NUMBER 4

000072 41 70 D 0C4 LA 7,V
000076 50 70 3 024 ST 7,36(0,3)
00007A 50 70 3 02C ST 7,44(0,3)
00007E 96 80 3 02C OI 44(3),X'80'
000082 41 10 3 024 LA 1,36(0,3)
000086 58 F0 3 014 L 15,A..IBMZXSA
00008A 05 EF BALR 14,15

* STATEMENT NUMBER 5

00008C 78 00 3 01C LE 0,28(0,3)
000090 70 00 D 0CA STE 0,A /*COMMONED ITEM*/

Figure 21. Use of Common Constants
Chapter 4. Run-Time Organization

This chapter tells you about the facilities of PL/I that are used at run-time and how they are organized. Topics covered are the task communications area, various registers, dynamic storage allocation, use of storage, load modules and their naming conventions and the multitasking library.

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

Dynamic Storage Allocation
The principles of the dynamic storage scheme are illustrated in Figure 22 on page 58.

The allocation and freeing of AUTOMATIC storage on a block-by-block basis implies a facility for the reuse of such storage. This technique of inter-block communications uses for each block a save area that contains register save information, AUTOMATIC variables, and housekeeping information.

This area is known as 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 of pointer values.

When a block is entered, the registers of the preceding block are stored in the previous DSA and a new DSA is acquired. A back-chain pointer to the previous DSA is placed in the new DSA. This arrangement allows access to information in previous blocks. Register 13 points to the head of the DSA for the current block. The code that carries out this and any other block initialization is known as the prolog code. To obviate the need for special coding in the main procedure, a dummy DSA is set up by an initialization routine. Register 13 points at this dummy DSA on entry to the main procedure.
1. The initial storage area (ISA) is acquired.
2. The program management area (a PL/I communications area) is placed at the head of the ISA.
3. All storage freed on a last in/first out basis (LIFO storage) is allocated at the low address end of the remaining unused storage.
4. When LIFO storage is freed, the most recently allocated element is the first to be freed. It is freed by being reabsorbed into the major free area.
5. Elements not freed on a last in/first out basis (non-LIFO storage) are allocated at the high address end of the free storage, or in a separate area known as HEAP.
6. When non-LIFO storage is freed, it is, where possible, absorbed into the major free area. Where this is not possible it is placed on a chain of free storage. The head of this chain is held at a fixed offset in the program management area. Areas on this chain are reused where possible.

Figure 22. Use of PL/I Dynamic Storage without Heap Storage
In addition to AUTOMATIC variables, certain other types of storage are allocated and freed dynamically. Items not freed on a last-in/first-out basis are kept in a second storage area called non-LIFO storage. If items within this area are freed, they are placed on a free-area chain. The storage scheme is handled partly by a compiled code and partly by a library routine.

In the LIFO storage stack, compiled code acquires and frees space.

HEAP storage is an area that is used for dynamically allocated storage, or controlled and dynamically based variables, in the program. This storage is separate from the ISA. It is controlled by the HEAP run-time option.

Contents of a Typical Load Module

The contents of a typical load module are shown in Figure 23 on page 60. The contents are:

- Object module (the executable machine instructions previously generated).
- Link edited routines. These routines include library routines, many of which are included in every executable program phase. These are the initialization routines. Other resident routines are included as required.

As well as the executable machine instructions, the program requires certain control information and addresses.

The Overall Use of Storage

Figure 24 on page 61 illustrates the overall use of storage. The program acquires an area known as the initial storage area (ISA) for program management and PL/I dynamic storage. The initialization routines set up the program management area which includes the TCA and the dummy DSA discussed above. PL/I dynamic storage allocations use the remainder of the ISA. The LIFO stack starts beyond the end of the program management area and expands, as necessary, toward the end of the ISA. Storage for I/O buffers and library routines is acquired by issuing GETMAIN macro instructions.

Non-LIFO dynamic storage starts at the end of the ISA and expands toward the LIFO stack. If you use HEAP storage, then separate storage areas are acquired.
Figure 23. Diagram of an Executable Program
### Figure 24. Use of Storage

<table>
<thead>
<tr>
<th>LOAD MODULE</th>
<th>NON-LIFO STORAGE</th>
<th>MAJOR FREE AREA</th>
<th>Initial Storage Area (ISA)</th>
<th>LAST-IN/FIRST-OUT (LIFO) STORAGE</th>
<th>PROGRAM MANAGEMENT AREA</th>
<th>Other Storage obtained by issuing GEMAIN macros</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiled code</td>
<td>Library modules</td>
<td>Addresses</td>
<td>Control blocks</td>
<td>Static variables</td>
<td>Dynamic storage</td>
<td>Storage for AUTOMATIC variables and compiler-generated temporaries, and other items allocated and freed on a block and procedure basis.</td>
</tr>
<tr>
<td>Other storage items</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Storage for transient library routines I/O buffers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Plus: Further allocation of dynamic storage</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Storage for CONTROLLED and dynamically allocated BASED variables.</td>
</tr>
</tbody>
</table>

- **TCA** (Task Communications Area) |
- **DSA** (Dynamic Storage Area) |
- Dummy DSA (Dynamic Storage Area) |
- Other Housekeeping Control Blocks |
- LAST-IN/FIRST-OUT (LIFO) STORAGE |
- MAJOR FREE AREA |
- NON-LIFO STORAGE |
- Initial Storage Area (ISA) |

**Chapter 4. Run-Time Organization**
Library Module Naming Conventions
PL/I library modules are named according to the conventions in this section. Usually the EBCDIC name is coded into the module close to its start. This allows the name of the routine to be easily seen in a dump.

The first three or four letters of the PL/I Library module name indicate where the module belongs. These initial letters and their locations follow.

<table>
<thead>
<tr>
<th>Initial Letter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBMB</td>
<td>PL/I Library</td>
</tr>
<tr>
<td>IBM0</td>
<td>Usually PLITEST but sometimes the PL/I Library</td>
</tr>
<tr>
<td>AQA</td>
<td>PLITEST</td>
</tr>
<tr>
<td>IBMF</td>
<td>PL/I Library, CICS support</td>
</tr>
<tr>
<td>IBMT</td>
<td>PL/I Library, Multitasking support</td>
</tr>
</tbody>
</table>

Note: Some CICS modules are link edited into the load module DFHSAP.

DUMP Eye Catchers and PLITEST
You will see module eye catchers in your dump. PL/I provides these to aid your problem diagnosis. Each eye catcher is a four-letter character string followed by the last date the module was compiled. The fourth letter of the module name and the mnemonic make up this character string.

For example, the eye catcher

B0PB 06/15/85

is for the module IBMOPBBA, which was last compiled on 06/15/85.

These same conventions are true in PLITEST.

The Multitasking Library
Two data sets hold the resident library modules, SYS1.PLIBASE and SYS1.PLTASK. SYS1.PLIBASE holds all modules needed to run non-multitasking programs. SYS1.PLTASK holds the multitasking versions of all modules that differ for multitasking and non-multitasking environments.

Both the 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 a multitasking or non-multitasking one. In order for multitasking programs to be link edited and run in a multitasking environment, the data set SY31.PLTASK must precede SYS1.PLIBASE in input to the linkage editor.
Chapter 5. Run-Time Problem Determination

This chapter contains the run-time problem determination chart. If you are just beginning your problem diagnosis, use the chart in Figure 26. Begin with the first block, Block 100, answer the question or perform the action specified, then go to the block indicated by the answer. Some blocks describe an action to be performed and also direct you to the next block.

If you have already made a preliminary diagnosis of your problem and you are familiar with PL/I, you can use the chart index below in Figure 25 to find the information you need.

<table>
<thead>
<tr>
<th>Run-Time Subject Covered</th>
<th>Block Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abend or Program Check</td>
<td>125</td>
</tr>
<tr>
<td>System Dump</td>
<td>126</td>
</tr>
<tr>
<td>Formatted PL/I Dump</td>
<td>129</td>
</tr>
<tr>
<td>Interrupt Type and Location</td>
<td>132</td>
</tr>
<tr>
<td>Source Statement Processed</td>
<td>139</td>
</tr>
<tr>
<td>Finding Registers</td>
<td>142</td>
</tr>
<tr>
<td>Search Argument Generation</td>
<td>143</td>
</tr>
<tr>
<td>Loop</td>
<td>144</td>
</tr>
<tr>
<td>Search Argument Generation</td>
<td>148</td>
</tr>
<tr>
<td>Wait</td>
<td>149</td>
</tr>
<tr>
<td>Unusual or Unexpected Output</td>
<td>150</td>
</tr>
<tr>
<td>Bad Output Data</td>
<td>154</td>
</tr>
<tr>
<td>Bad File Record</td>
<td>156</td>
</tr>
<tr>
<td>Search Argument Generation</td>
<td>160</td>
</tr>
<tr>
<td>Performance</td>
<td>161</td>
</tr>
</tbody>
</table>

Figure 25. Run-Time Problem Determination Index

<table>
<thead>
<tr>
<th>Block No.</th>
<th>Question</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Is this an abend?</td>
<td>Yes 106</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 101</td>
</tr>
<tr>
<td>101</td>
<td>Is this a problem relating to a message?</td>
<td>Yes 106</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 102</td>
</tr>
<tr>
<td>102</td>
<td>Is this a loop?</td>
<td>Yes 106</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 103</td>
</tr>
<tr>
<td>103</td>
<td>Is this a wait?</td>
<td>Yes 106</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 104</td>
</tr>
<tr>
<td>104</td>
<td>Is this unusual or unexpected output?</td>
<td>Yes 106</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 105</td>
</tr>
<tr>
<td>105</td>
<td>Is this a performance problem?</td>
<td>Yes 106</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 124</td>
</tr>
</tbody>
</table>

Figure 26 (Part 1 of 11). Run-Time Problem Determination Chart
<table>
<thead>
<tr>
<th>Block No.</th>
<th>Question</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>106</td>
<td>Add to the program the following block:</td>
<td>Yes 107&lt;br&gt;No 109</td>
</tr>
<tr>
<td></td>
<td>ON ERROR BEGIN;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ON ERROR SNAP SYSTEM;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PUT DATA;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>END;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Recompile and rerun the program. If the problem is solved, END. If the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>problem still exists: has the program ever worked before?</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>Has anything in the environment changed?</td>
<td>Yes 111&lt;br&gt;No 108</td>
</tr>
<tr>
<td></td>
<td>Look for source code changes, system control program (SCP) changes,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PTFs, release fixes, etc.</td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>Is the entry from:</td>
<td>Yes 125&lt;br&gt;Yes 144&lt;br&gt;Yes 149&lt;br&gt;Yes 150&lt;br&gt;Yes 161</td>
</tr>
<tr>
<td></td>
<td>MESSAGE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LOOP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WAIT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UNUSUAL or UNEXPECTED OUTPUT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PERFORMANCE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Note: If you are here via the &quot;environment changed&quot; route, follow the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>major symptom code being experienced.</td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>Has the source code been checked for accuracy and all &quot;E&quot; or &quot;S&quot; level</td>
<td>Yes 108&lt;br&gt;No 110</td>
</tr>
<tr>
<td></td>
<td>compiler messages corrected?</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>Correct the conditions causing the &quot;E&quot; or &quot;S&quot; level messages. Recompile</td>
<td>Yes END&lt;br&gt;No 120</td>
</tr>
<tr>
<td></td>
<td>and test the program. Is the problem solved? Note: If the problem is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>solved, but you feel the message was generated in error, follow the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;NO&quot; path.</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>Has the source code of the program been changed?</td>
<td>Yes 115&lt;br&gt;No 112</td>
</tr>
<tr>
<td>112</td>
<td>Has any maintenance been applied to the PL/I compiler and/or libraries?</td>
<td>Yes 116&lt;br&gt;No 113</td>
</tr>
<tr>
<td>113</td>
<td>Has the PL/I compiler release level changed?</td>
<td>Yes 118&lt;br&gt;No 114</td>
</tr>
</tbody>
</table>

Figure 26 (Part 2 of 11). Run-Time Problem Determination Chart
<table>
<thead>
<tr>
<th>Block No.</th>
<th>Question</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>114</td>
<td>You probably do not have a PL/I problem. In getting to this block, you have indicated that the program has compiled and run successfully before. Now you should investigate the system control program for changes that could have affected this program. <strong>Note:</strong> System control program refers to the operating system. If you still believe this to be a PL/I problem, go to Block 108 and follow the major symptom code.</td>
<td>See Note</td>
</tr>
<tr>
<td>115</td>
<td>Were there any messages at compile time that could have caused this problem? If so, correct those conditions. Recompile and test the program. Is the problem solved?</td>
<td>Yes END No 120</td>
</tr>
<tr>
<td>116</td>
<td>Is(Are) the fix(es) or PTF(s) installed correctly? <strong>Note:</strong> Check early warning microfiche or have the IBM Support Center check SSF for errors on all PTFs and fixes applied. Use the keyword PExxxxx for PTF errors.</td>
<td>Yes 121 No 117</td>
</tr>
<tr>
<td>117</td>
<td>Reinstall the PTF or fix correctly and test. Is the problem solved?</td>
<td>Yes END No 120</td>
</tr>
<tr>
<td>118</td>
<td>Is the release level installed correctly? <strong>Note:</strong> Search early warning microfiche or have the IBM Support Center search SSF for any PTFs and errors applicable to this release. Are there any messages from the link edit steps?</td>
<td>Yes 121 No 119</td>
</tr>
<tr>
<td>119</td>
<td>Reinstall the release level correctly, plus any PTFs or fixes that are applicable, and test. Is the problem solved?</td>
<td>Yes END No 120</td>
</tr>
<tr>
<td>120</td>
<td>Have the symptoms changed?</td>
<td>Yes 100 No 121</td>
</tr>
<tr>
<td>121</td>
<td>Search early warning microfiche or have the IBM Support Center do an SSF search. For more information about doing an SSF search, see Chapter 7, &quot;Using SSF and CSSF Search Arguments&quot; on page 113. Any hits?</td>
<td>Yes 122 No 108</td>
</tr>
<tr>
<td>122</td>
<td>Apply applicable fix(es) from hit(s) and test. Is the problem solved?</td>
<td>Yes END No 123</td>
</tr>
<tr>
<td>123</td>
<td>Have the symptoms changed?</td>
<td>Yes 100 No 108</td>
</tr>
<tr>
<td>124</td>
<td>Something has apparently been overlooked. A failure must have occurred and it must have been one of the previously mentioned types. Review the symptoms again. If the problem does not fit any of the stated symptoms, go to Block 164.</td>
<td>164</td>
</tr>
</tbody>
</table>

Figure 26 (Part 3 of 11). Run-Time Problem Determination Chart
<table>
<thead>
<tr>
<th>Block No.</th>
<th>Question</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>125</td>
<td>Do you have a system dump rather than a PL/I formatted dump?</td>
<td>Yes 126 No 129</td>
</tr>
<tr>
<td></td>
<td>Review the section on run-time dump analysis that precedes these charts.</td>
<td></td>
</tr>
<tr>
<td>126</td>
<td>To reduce the possibility of a user error, rerun the program with the SIZE, SUBSCRIPTRANGE, STRINGSIZE, and STRINGRANGE conditions enabled. Were any of these conditions raised?</td>
<td>Yes 127 No 129</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> For instructions, see OS PL/I Version 2 Programming: Language Reference.</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>This probably means you have a programming problem. Correct the code that caused the condition, and rerun the program. Is the problem solved?</td>
<td>Yes END No 128</td>
</tr>
<tr>
<td>128</td>
<td>Have the symptoms changed?</td>
<td>Yes 100 No 129</td>
</tr>
<tr>
<td>129</td>
<td>Do you have an IBMxxxx message?</td>
<td>Yes 130 No 131</td>
</tr>
<tr>
<td>130</td>
<td>The IBMxxxx message will give you the following information:</td>
<td>131</td>
</tr>
<tr>
<td></td>
<td>Message number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ON-code number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Condition raised (if applicable)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Statement number being processed (if “GOSTMT” was specified)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Offset of statement being processed (You can use the table of offsets to find the statement number from this.)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Procedure name being run</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Go to Block 131.</td>
<td></td>
</tr>
<tr>
<td>131</td>
<td>Do you already know the type and location of the interrupt?</td>
<td>Yes 142 No 132</td>
</tr>
<tr>
<td>132</td>
<td>Do you have a system dump rather than a PL/I formatted dump?</td>
<td>Yes 136 No 133</td>
</tr>
<tr>
<td>133</td>
<td>Does the PL/I dump have trace information?</td>
<td>Yes 134 No 135</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> Find the words “<strong><strong>CALLING TRACE</strong></strong>”.</td>
<td></td>
</tr>
</tbody>
</table>

Figure 26 (Part 4 of 11). Run-Time Problem Determination Chart
<table>
<thead>
<tr>
<th>Block No.</th>
<th>Question</th>
<th>Action</th>
</tr>
</thead>
</table>
| 134      | To find the type of interrupt:  
1. For an example of trace information in a dump, see "Housekeeping Information in All Dumps" on page 92.  
2. The ON-code (1) type of ON-unit, (2) and reason for entry, (3) as well as who called the error handler (4) are given in the trace information.  
3. The ON-code is your type of interrupt.  
To find the interrupt locations:  
1. Find IBMBERR's DSA (see "Housekeeping Information in All Dumps" on page 92).  
2. Chain back one DSA (X'04' in each DSA holds the address of the previous DSA).  
3. The address of the interrupt is at X'0C' in this DSA.  
To find the register contents on interrupt:  
1. If the interrupt was an abend, there will be no register information available unless SPIE and STAE were in effect.  
2. If the interrupt was a software detected interrupt, the contents of registers 14 through 11 at the time of interrupt are at offset X'08' in the previous DSA.  
3. If the interrupt was a program check, registers 0 through 11 are at offset X'14' in the previous DSA, and registers 14 and 15 are at X'5C' in the DSA for IBMBERR.  
Go to Block 137.  
Note: IBMBERR could be IBMFERR. | 137 |

Figure 26 (Part 5 of 11). Run-Time Problem Determination Chart
<table>
<thead>
<tr>
<th>Block No.</th>
<th>Question</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>135</td>
<td>To find the type of interrupt: find and interpret the on communication area (ONCA) which will give you the ON-code.</td>
<td>136</td>
</tr>
<tr>
<td></td>
<td>1. Find IBMBERR's DSA. (Register 13 has the current DSA address. Offset X'04' in each DSA holds the address of the previous DSA. IBMBERR's DSA has X'EEEE' in the second halfword of the first word.)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. Locate the halfword at offset X'54' of this DSA, which contains the error code. If this error code is X'0C' or X'0D' then locate the address of the LWS at offset X'48' from IBMBERR's DSA and locate the pointer to the current ONCA. (Offset X'02' in the LWS.)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. From the current ONCA, chain back to the previous ONCA (the chain back field is the first fullword of each ONCA).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. Locate the 2-byte error code at X'04'.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5. Using Figure 32 on page 86, find the base number associated with the first byte of the error code.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6. Translate the right-hand five bits (of the second byte of the error code) to decimal.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7. Add this value to the base number to get the ON-code.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8. Look up the ON-code in the OS PL/I Version 2 Programming: Language Reference. If you have a system dump, go to Block 136. Note: IBMBERR could be IBMFERR.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>135 (cont'd)</th>
<th>To find the interrupt location:</th>
<th>137</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1. Register 13 holds current DSA address.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2. Offset X'04' in each DSA holds address of previous DSA.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. Chain back to the DSA before the error handler's DSA.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. The interrupt address is at offset X'0C' in this DSA. Go to Block 137.</td>
<td></td>
</tr>
</tbody>
</table>

| 136         | If possible, obtain a PL/I dump, then go to block 133. If you cannot obtain a PL/I dump but you do have a system dump, you will find the interrupt address in the second word of the PSW in the system dump. Go to Block 137. | 133, 137 |

Figure 26 (Part 6 of 11). Run-Time Problem Determination Chart
<table>
<thead>
<tr>
<th>Block No.</th>
<th>Question</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>137</td>
<td>Compare the interrupt address to the link map.</td>
<td>138</td>
</tr>
<tr>
<td></td>
<td>Is the address within a library module or outside the link map?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>If the address is outside the link map, the interrupt probably occurred in</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a transient library module.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>If the address is within the link map, the interrupt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>occurred in a resident library module.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Remember the module name for your early warning microfiche or SSF search.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Go to Block 138.</td>
<td></td>
</tr>
<tr>
<td>138</td>
<td>Now you know the type of the interrupt and, if you have</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>a message or a PL/I dump with trace information, you should also know</td>
<td></td>
</tr>
<tr>
<td></td>
<td>what source statement was being run.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Go to Block 142.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>If you have a system dump, or no trace information, or for some other</td>
<td>139</td>
</tr>
<tr>
<td></td>
<td>reason you do not know the source statement being run, go to Block 139.</td>
<td></td>
</tr>
<tr>
<td>139</td>
<td>LIST option in effect:</td>
<td>Yes 140</td>
</tr>
<tr>
<td></td>
<td>OFFSET option in effect.</td>
<td>Yes 141</td>
</tr>
</tbody>
</table>

Figure 26 (Part 7 of 11). Run-Time Problem Determination Chart
<table>
<thead>
<tr>
<th>Block No.</th>
<th>Question</th>
<th>Action</th>
</tr>
</thead>
</table>
| 140      | 1. Locate the error handler DSA. (Register 13 has the current DSA address. X'04' in each DSA holds the address of the previous DSA. IBMBERR's DSA has X'EEEE' in the first word.)  
2. Chain back until a procedure DSA is found. (Procedure DSAs have bits 4 and 5 set to '00'B.)  
3. Chain back one more DSA.  
   Note: IBMBERR could be IBMFERR.  
4. Note the address held at offset X'10' (register 15 for the procedure in which the interrupt occurred).  
5. Subtract this entry point address from the interrupt address to obtain an offset. If the interrupt occurred in a library module, subtract the entry point address from the entry point address of the procedure (Register 14).  
6. This offset should then be added to the offset of the REAL ENTRY of the procedure in which the interrupt occurred as given in the object listing. (For main procedures, this will be the length of the procedure name plus one, rounded up to the next multiple of four.)  
7. In the object listing, this offset points to the assembler instruction after the instruction that caused the interrupt. The PL/I statement number is given in the listing preceding the code with which it is associated. Sometimes the statement statement number may not be exact, if the OPT (TIME) option is specified. Go to Block 142. | 142    |
| 141      | 1. Obtain the offset address of the interrupt as described in steps 1 through 5 in Block 140.  
2. Use this offset with the offset and Statement Table in the listing to find the statement number. Go to Block 142. | 142    |
| 142      | If your ON-code was 8094, 8095, or 8096, you may want to know the contents of the registers at the time of the program check. Refer to Chapter 6, "Debugging Using Dumps" on page 75 for more information.  
   Go to Block 148 if your major symptom code is loop. Otherwise, go to Block 143. | 148    |
<p>|          |                                                                                                                                                                                                  | 143    |</p>
<table>
<thead>
<tr>
<th>Block No.</th>
<th>Question</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>143</td>
<td>Search early warning microfiche or have the IBM Support Center do an SSF search. More information doing an SSF searches is in Chapter 7, &quot;Using SSF and CSSF Search Arguments&quot; on page 113. Any hits?</td>
<td>Yes 162 No 164</td>
</tr>
<tr>
<td>144</td>
<td>If a loop appears to be occurring, use the instruction step mode, if possible, to capture all, or at least part, of the loop addresses. Then cancel the job with a dump. Is the loop in the user program area? Note: Use the link map to determine this.</td>
<td>Yes 145 No 146</td>
</tr>
<tr>
<td>145</td>
<td>If the loop is within the user program, it is probably a program logic error. Correct the situation and rerun the program. If you still believe the problem to be in PL/I, go to Block 148.</td>
<td>148</td>
</tr>
<tr>
<td>146</td>
<td>Find the module name(s) in which the loop occurs. Use the link map to determine this. If the loop addresses are outside the link map, the loop is probably occurring in a transient library module. If it is in the link map, remember the name. Note: The control blocks portion of an ABDUMP (SNAP) contains information on the entry point address and length of transients loaded. Go to Block 147.</td>
<td>147</td>
</tr>
<tr>
<td>147</td>
<td>You may want to know which source statement is being processed. Go to Block 138.</td>
<td>138</td>
</tr>
<tr>
<td>148</td>
<td>Search early warning microfiche or have the IBM Support Center do an SSF search using:</td>
<td>Yes 162 No 164</td>
</tr>
<tr>
<td></td>
<td>• component-id 5668909</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• EXEC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• LOOP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• module name(s) (if applicable)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• type of statement being processed (if applicable—search without it first).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Any hits?</td>
<td></td>
</tr>
<tr>
<td>149</td>
<td>Usually wait states are caused by the system control program, not by PL/I. Get a dump and determine what is being waited for. To do this, locate the ECB/CCB pointed to by register 1. If you still believe the problem is in PL/I, search early warning microfiche or have the IBM Support Center do an SSF search, using:</td>
<td>Yes 162 No 164</td>
</tr>
<tr>
<td></td>
<td>• component-id 5668909</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• EXEC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• WAIT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Any hits?</td>
<td></td>
</tr>
</tbody>
</table>

Figure 26 (Part 9 of 11). Run-Time Problem Determination Chart
<table>
<thead>
<tr>
<th>Block No.</th>
<th>Question</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td>Rerun the program with the following conditions enabled: SIZE, SUBSCRIPTRANGE, STRINGSIZE, STRINGRANGE. Were any of these conditions raised?</td>
<td>Yes 151 No 153</td>
</tr>
<tr>
<td>151</td>
<td>Correct the code that raised these conditions and rerun the program. Is the problem solved?</td>
<td>Yes END No 152</td>
</tr>
<tr>
<td>152</td>
<td>Have the symptoms changed?</td>
<td>Yes 100 No 153</td>
</tr>
<tr>
<td>153</td>
<td>If you do not have a dump, you may want to get one by calling PLIDUMP at appropriate places in the program. See Chapter 6, &quot;Debugging Using Dumps&quot; on page 75 for more information. Is your problem due to bad output data, such as on an output report?</td>
<td>Yes 154 No 155</td>
</tr>
<tr>
<td>154</td>
<td>You need to look at your variable fields in the dump. First, determine the types of variables you are working with. They will be one of the following types: static, automatic, controlled, based, area. &quot;Finding Variables&quot; on page 107 explains how to find these variables in a dump. Go to Block 155.</td>
<td>155</td>
</tr>
<tr>
<td>155</td>
<td>Is your problem related to a bad file record?</td>
<td>Yes 156 No 160</td>
</tr>
<tr>
<td>156</td>
<td>You need to look at the file information in the dump. Do you have a PL/I dump with file information formatted?</td>
<td>Yes 157 No 158</td>
</tr>
<tr>
<td>157</td>
<td>The FCB, ENVB, DCB, and for VSAM the IOCBO and ACB will be formatted for each open file. Go to Block 159.</td>
<td>159</td>
</tr>
<tr>
<td>158</td>
<td>An offset from register 3 points to the DCLCB. The first word in the DCLCB points to the FCB. The offset from register 3 can be found in the static storage map in the compile listing. The maps of the file control blocks are in Appendix A, &quot;Control Blocks&quot; on page 119. Go to Block 159.</td>
<td>159</td>
</tr>
<tr>
<td>159</td>
<td>Check the file attributes in the dump with those in the source program to be sure they are the same. One common error in overlay programs is not specifying the exact attributes in an overlay for a file as were specified in the root. Go to Block 160.</td>
<td>160</td>
</tr>
</tbody>
</table>

Figure 26 (Part 10 of 11). Run-Time Problem Determination Chart
<table>
<thead>
<tr>
<th>Block No.</th>
<th>Question</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>160</td>
<td>Search early warning microfiche or have the IBM Support Center do an SSF search. Information about doing an SSF searches is in Chapter 7, “Using SSF and CSSF Search Arguments” on page 113. Any hits?</td>
<td>Yes 162</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No 164</td>
</tr>
</tbody>
</table>
| 161      | Performance problems usually show up after some environment change or when using some function you have not used before. Try to identify this. Search early warning microfiche or have the IBM Support Center do an SFF search using:  
  * component-id 5668909  
  * EXEC  
  * PERFM.  
  If you can, also include a description of the action being performed. Any hits? | Yes 162      |
|          |                                                                          | No 164       |
| 162      | Apply the fix(es) from SSF and test. Is the problem solved?              | Yes END      |
|          |                                                                          | No 163       |
| 163      | Have the symptoms changed?                                               | Yes 100      |
|          |                                                                          | No 164       |
| 164      | Contact the IBM Support Center for assistance.  
  Have available the following documentation:  
  * Compilation listing with LIST and MAP options specified  
  * Job control statements  
  * Linkage Editor map  
  * Run-time dump (if applicable) |               |

Figure 26 (Part 11 of 11). Run-Time Problem Determination Chart
Chapter 6. Debugging Using Dumps

The OS PL/I Optimizing Compiler allows you to obtain a run-time dump by calling PLIDUMP. Using SYSABEND or SYSUDUMP in the JCL does not normally result in a dump after a program interrupt or, except in certain exceptional cases, after an ABEND. This is because SPIE/ESPIE and STAE/ESTAE 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 statement was included in the JCL, an ABEND dump will then be generated.

ABEND dumps are possible under these 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.
4. An error occurred in the program and the user has coded the User Exit module.

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.

It is always possible for the programmer to ask an operator to request a stand-alone dump at any point in the program. The need to do this should, however, occur only infrequently.

For information about compiled code register usage and library register usage see Figure 11 on page 24 and Figure 12 on page 26.

Considerations

A DD statement with the ddname PLIDUMP or PL1DUMP, a FILEDEF command in CMS, or an ALLOCATE command in TSO must be supplied to define the data set for the dump.

The data set defined by the PLIDUMP DD statement must have DSORG=PS specified or assumed by default, and must have one of the following attributes:
- Correct DCB parameters for a system SNAP dump
- Allocated to disk
- Allocated to SYSOUT
- Allocated to the terminal or unit-record device.

To get a formatted dump, ON ERROR must CALL PLIDUMP.
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 Chapter 3, "Compiler Output" is assumed. A summary of how to use this chapter when debugging is given in Figure 27 on page 77.

This chapter is divided into seven sections:

• "Section 1: How to Obtain a PL/I Dump".
  This section 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: Suggested Debugging Procedures".
  This section describes two ways of debugging a PL/I program using a dump. The first shows a PL/I dump that was called from an ERROR ON-unit; the second shows debugging with a system dump that was probably generated because the housekeeping control blocks were overwritten.

• "Section 3: Locating Specific Information".
  This section describes how to find various data areas and other information. It is indexed and numbered for quick reference.

• "Section 4: Special Considerations for Multitasking".
  This section describes the special considerations that must be taken into account when debugging a program that uses multitasking.

• "Section 5: Special Considerations for CICS".
  This small section tells you about the differences between the PL/I-CICS run-time environment and the PL/I batch and multitasking environments.

• "Section 6: User Exit Considerations".
  This section describes the user exit and how to use it.

• "Section 7: SYSTEM Option Considerations".
  This section gives you some information about the SYSTEM option and refers you to the OS PL/I Version 2 Programming Guide for more details.
Follow the most suitable check list in section 2 of this chapter. Refer to keyed items in section 3 of this chapter for details.

Figure 27. How to Use this Chapter When Debugging

If you are familiar with system dump methods, read "Section 1: How to Obtain a PL/I Dump" on page 78 before requesting a dump. PL/I uses methods that do not follow OS. Use the next two sections when debugging. If you know what
not follow OS. Use the next two sections when debugging. If you know what you are looking for, go directly to "Contents" on page 92. This section directs you to numbered sections that give details of how to find particular items. If you have no preferred scheme of your own, you can follow the recommended procedures in “Section 2: Suggested Debugging Procedures” on page 87. It cross-references to the items in “Section 3: Locating Specific Information,” so that the details of the steps involved can be quickly found.

Section 1: How to Obtain a PL/I Dump

In order to get a formatted PL/I dump, you must include a call to PLIDUMP in your program.

CALL PLIDUMP

The statement CALL PLIDUMP may appear wherever a CALL statement is 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 continues after the dump.

H  Hexadecimal. A SNAP hexadecimal dump of the region 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.

K  Produce a hexadecimal dump of the TIOAS and TWA (CICS control blocks) if they exist.

NK  No dump of CICS blocks.
Tasking Options

A  All, which results in a dump of all active tasks including the control task.

O  Only, which results in a dump of the current task and a dump of the control task.

E  Exit, which 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 is output.

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 occasion. This simplifies identification of the point at which the dump was called.

Suggested 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 28 on page 80.
These options give compiled code listing and static storage map, essential for interpreting any dump. MAP results in the generation of a table showing offsets of static and automatic variables from their defining bases.

Provides trace of last n branch-out/branch-in points in up to m blocks if SNAP or PLIDUMP with trace is used.

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);
```

- **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
- **K** Produce a hexadecimal dump of TIOAS and TWA
- **NK** No dump of CICS blocks

Printed at head of dump. May be up to 90 characters long.

Figure 28. Code for Debugging
Avoiding Recompilation

If an ERROR ON-unit containing a call to PLIDUMP is to be included in an existing program, it is necessary to recompile the program. This course is advisable as it allows other diagnostic aids, such as SUBSCRIPTRANGE, to be included. However, if recompilation 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 29. 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.

Note: You can call PLIDUMP without recompiling your program by using PLITEST. You can use the run-time TEST option to have PLITEST easily gain control of your program. Once PLITEST has control, it can do a number of things, including issuing a CALL PLIDUMP. For more information about this method of calling PLIDUMP, see OS PL/I Version 2 Programming: Using PLITEST.

```
BOOTSTRAP: PROC OPTIONS (MAIN);
   DCL program* ENTRY EXTERNAL;
   ON ERROR CALL PLIDUMP ('HB', 'BOOTSTRAP');
   CALL program*;
END;
```

Figure 29. Suggested Method of Obtaining a Dump when Recompilation is Particularly Undesirable

Note to Figure 29:

* Insert the name of the program to be dumped at the points marked "program*" in this example.

If the program that requires dumping already has an ERROR ON-unit, this will override the ERROR ON-unit in the bootstrap program. However, if you are using PLITEST, PLITEST gains control prior to the ON-units in the program.

In certain circumstances, a dump can still be obtained.

1. If the reason for the entry to the ON-unit is the occurrence of a PL/I condition, an ON-unit for this condition in the bootstrap program will result in a dump before the ERROR ON-unit is run.
(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.

IBM does not recommend the bootstrap method unless you have particularly strong reasons for avoiding recompilation.

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 30 on page 84. The contents of particular sections follow.

Headings
The dump is headed by the line

***PL/I DUMP***

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. 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.

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 (for example, 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 3- or 4-letter abbreviation. A list of these abbreviations is given in Figure 31 on page 84.

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, the following information is also included:
   
a. The address of IBMBERRs 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.
**USER IDENTIFIER**: EXAMPLE OF PLIDUMP

**CALLING TRACE**

PLIDUMP WAS CALLED FROM STATEMENT NUMBER 2 AT OFFSET +0000BA FROM A ERR TYPE ON-UNIT WITH ENTRY ADDRESS 02200E58

(AID DSA ADDRESS 0000C268)

**ERROR DIAGNOSTICS**

**PL/I CONDITION DETECTED**: CONV

OMCODE = E12

ONCHAR = ‘T’

SEE LANGUAGE REFERENCE MANUAL

ONSOURCE = THIS WILL RAISE CONVERSION ERROR

E=3

AS ABOVE IN HEXADECIMAL

ADDRESS OF ERROR HANDLER’S SAVE AREA 0000C888

REGISTERS ON ENTRY TO ERROR HANDLER

REG 0-7 0000C880 0000C880 0000C800 02200E7E 0000C550 0000C7A0 0000C818 02200CBA

REGS 8-15 00000001 000007A3 00000000 0220264A 0000C910 0000C830 02202D7C 0000C4C4

**END OF ERROR DIAGNOSTICS**

WHICH WAS CALLED FROM A LIBRARY MODULE WITH ENTRY ADDRESS 02200E58 (AND DSA ADDRESS 0000C39)

WHICH WAS CALLED FROM A LIBRARY MODULE WITH ENTRY ADDRESS 02200E3E (AND DSA ADDRESS 0000C40)

WHICH WAS CALLED FROM A LIBRARY MODULE WITH ENTRY ADDRESS 02200F68 (AND DSA ADDRESS 0000C78)

WHICH WAS CALLED FROM STATEMENT NUMBER 6 AT OFFSET +0000BA FROM PROCEDURE EXAMPLE WITH ENTRY ADDRESS 02200E98

(AID DSA ADDRESS 0000C588)

**TRACE OF PL/I CONTROL BLOCKS**

**ERROR ADDRESS**

OFFSET 0 4 8 10 14 18 1C

00000C10 00000E90 00000C88 00000E90 00000C80 00000C80 00000C80 00000C80 00000C80

00000C80 00000C80 00000C80 00000C80 00000C80 00000C80 00000C80 00000C80 00000C80

**END OF CALLING TRACE**

**TCA IMPLEMENTATION APPENDAGE**

**LIBRARY WORK SPACE**

**CONTENTS OF REGISTER SAVE AREA**

REGS 0-7 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

REGS 8-15 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

**DYNAMIC SAVE AREA (ON-UNIT)**

**CONTENTS OF REGISTER SAVE AREA**

Figure 30. An Example of PLIDUMP
Abbreviation | Condition Name
--- | ---
AREA | AREA
CHECK | 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 31. Abbreviations for Condition Names Used in PLIDUMP Trace Information

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.

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.

### Figure 32. Error Code Field Lookup Table

<table>
<thead>
<tr>
<th>Byte 1</th>
<th>PL/I Condition, If Any</th>
<th>Base No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'02'</td>
<td>ZERODIVIDE</td>
<td>320</td>
</tr>
<tr>
<td>X'03'</td>
<td>FIXEDOVERFLOW</td>
<td>310</td>
</tr>
<tr>
<td>X'04'</td>
<td>SIZE</td>
<td>340</td>
</tr>
<tr>
<td>X'05'</td>
<td>CONVERSION</td>
<td>600</td>
</tr>
<tr>
<td>X'06'</td>
<td>OVERFLOW</td>
<td>300</td>
</tr>
<tr>
<td>X'07'</td>
<td>UNDERFLOW</td>
<td>330</td>
</tr>
<tr>
<td>X'08'</td>
<td>STRINGSIZE</td>
<td>150</td>
</tr>
<tr>
<td>X'09'</td>
<td>STRINGRANGE</td>
<td>350</td>
</tr>
<tr>
<td>X'0A'</td>
<td>SUBSCRIPTRANGE</td>
<td>520</td>
</tr>
<tr>
<td>X'0B'</td>
<td>AREA</td>
<td>360</td>
</tr>
<tr>
<td>X'0C'</td>
<td>ERROR</td>
<td>009</td>
</tr>
<tr>
<td>X'0D'</td>
<td>FINISH</td>
<td>004</td>
</tr>
<tr>
<td>X'0E'</td>
<td>CHECK</td>
<td>510</td>
</tr>
<tr>
<td>X'0F'</td>
<td>CONDITION</td>
<td>500</td>
</tr>
<tr>
<td>X'10'</td>
<td>KEY</td>
<td>050</td>
</tr>
<tr>
<td>X'11'</td>
<td>RECORD</td>
<td>020</td>
</tr>
<tr>
<td>X'12'</td>
<td>UNDEFINEDFILE</td>
<td>080</td>
</tr>
<tr>
<td>X'13'</td>
<td>ENDFILE</td>
<td>070</td>
</tr>
<tr>
<td>X'14'</td>
<td>TRANSMIT</td>
<td>040</td>
</tr>
<tr>
<td>X'15'</td>
<td>NAME</td>
<td>010</td>
</tr>
<tr>
<td>X'16'</td>
<td>ENDPAGE</td>
<td>090</td>
</tr>
<tr>
<td>X'17'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'18'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'19'</td>
<td>PENDING</td>
<td>100</td>
</tr>
<tr>
<td>X'1A'</td>
<td>ATTENTION</td>
<td>400</td>
</tr>
<tr>
<td>X'1B'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'1C'</td>
<td>ERROR</td>
<td>9250</td>
</tr>
<tr>
<td>X'1D'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'1E'</td>
<td>ERROR</td>
<td>9200</td>
</tr>
<tr>
<td>X'1F'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X'20'</td>
<td></td>
<td>3500</td>
</tr>
<tr>
<td>X'21'</td>
<td></td>
<td>4050</td>
</tr>
<tr>
<td>X'22'</td>
<td></td>
<td>5050</td>
</tr>
<tr>
<td>X'23'</td>
<td></td>
<td>5000</td>
</tr>
<tr>
<td>X'24'</td>
<td>ERROR</td>
<td>9050</td>
</tr>
<tr>
<td>X'25'</td>
<td>ERROR</td>
<td>1000</td>
</tr>
<tr>
<td>X'26'</td>
<td></td>
<td>4000</td>
</tr>
<tr>
<td>X'27'</td>
<td></td>
<td>xxxx</td>
</tr>
<tr>
<td>X'28'</td>
<td></td>
<td>4050</td>
</tr>
<tr>
<td>X'29'</td>
<td>ERROR</td>
<td>0003</td>
</tr>
<tr>
<td>X'2A'</td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td>X'2B'</td>
<td>ERROR</td>
<td>1550</td>
</tr>
<tr>
<td>X'2C'</td>
<td></td>
<td>1500</td>
</tr>
<tr>
<td>X'2D'</td>
<td></td>
<td>2000</td>
</tr>
<tr>
<td>X'2E'</td>
<td></td>
<td>3768</td>
</tr>
<tr>
<td>X'2F'</td>
<td>ERROR</td>
<td>3000</td>
</tr>
<tr>
<td>X'30'</td>
<td>ERROR</td>
<td>3800</td>
</tr>
<tr>
<td>X'31'</td>
<td></td>
<td>3900</td>
</tr>
<tr>
<td>X'32'</td>
<td></td>
<td>9000</td>
</tr>
<tr>
<td>X'33'</td>
<td></td>
<td>8090</td>
</tr>
</tbody>
</table>

---

86 OS PL/I Version 2 Problem Determination

LY27-9528-0 © Copyright IBM Corp. 1985, 1987
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; that is, 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 are also printed.

If the option A(all) is used in a multitasking program, 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: Suggested 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 a dump was requested after an error).
2. The type of error (if a dump was requested after an error).
3. The values in the general registers when the dump was requested 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 a 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: Locating Specific Information" on page 92, refer to the key areas shown above.

The object program listing allows you to study the instructions that are being carried out and to find various control blocks in static storage. The linkage editor map allows you to identify particular parts of the executable program phase and to identify the routine associated with each DSA. The object program listing is produced by the LIST compiler option; the linkage editor map, by MAP.

**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. Instead, look at the trace information.

### Debugging Overlaid Storage

Storage overlay is one of the most common errors you usually debug with a dump. In PL/I applications, overlay problems can be divided into these categories:

- Are you using a subscript outside the declared bounds (SUBSCRIptrANGE)?
- Did you attempt to assign a string to a target with a shorter maximum length (STRINGSIZE)?
- Does one of the arguments to a SUBSTR reference fail to comply with the rules described for the SUBSTR built-in function (STRINGRANGE)?
- Were significant high-order (left-most) binary or decimal digits lost during an assignment to a variable, an intermediate result, or on an input/output operation (SIZE)?
- Are you reading a variable-length file into a variable?
- Are you using a pointer variable?

By understanding these problem areas before you proceed through the dump, you can isolate the problem much faster.

The first four categories are associated with the indicated PL/I conditions, all of which are disabled by default. If you suspect one of these problems is in your program, use a condition prefix on the suspected statement or use the condition prefix on the BEGIN or PROCEDURE statement that defines the block containing the suspected statement.

The fifth category occurs when you read a data record into a variable that is too small. This type of problem only happens with variable-length files, and can often be isolated by examining the data in the file information, and the data in the buffer.

The last category occurs when you misuse a pointer variable. This type of storage overlay is particularly difficult to isolate.

There are a number of ways pointer variables can be misused:
1. When a READ statement with the SET option is executed, a value is placed in a pointer. If you then execute a WRITE statement or another READ SET option with another pointer, you will overlay your storage if you try to use the original pointer.

2. When you attempt to use a pointer to allocated storage that has already been freed, you can also cause a storage overlay.

3. When you attempt to use a pointer, set with the ADDR built-in function, as a base for data with different attributes, you can cause a storage overlay.

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:

1. When PLIDUMP has been called from an ERROR or other ON-unit
2. When only a system ABEND dump has been generated.

Other possible situations are when you request a dump a specified point in the program, or when you request a stand-alone dump. 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: Locating Specific Information” on page 92.

Throughout each of the two recommended procedures given in the following paragraphs, there are cross-references to the methods given in “Section 3: Locating Specific Information.” The cross-references consist of the keys by which the methods are identified; for example, H6, D5. These keys are listed in “Housekeeping Information in All Dumps” on page 92.

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 is available at the head of the dump. An error message is generated, which provides a useful starting point. First, examine the type of the error and the point where it occurs. Next, examine the ONCODE and other condition built-in function values, along with the trace information. We suggest the following procedure:

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 codes are given in the OS PL/I Version 2 Programming: Language Reference.)

2. Find the location of the error (P1 on page 92) and the block in which the error occurred (H12 on page 93). If the error occurred in a library module, see H14 on page 93. This information is normally available from the head of the PLIDUMP in the trace information.

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 and 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 the 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 the logic of code being executed from object listing.

System ABEND Dump

Provided a SYSABEND or a SYSUDUMP card is included in the JCL, a system 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/ESTAE 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 results in a dump.

Thus, an ABEND is produced if:

- The program interrupt exit was reset during the program.
  This exit is normally set by the program initialization routines to prevent a dump.
- The program interrupt exit was never set at all. This possibility is extremely unlikely.
- The program check exit itself is not working, and the SPIE/ESPIE 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 was 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.

If an interrupt occurs during error handling, an ABEND with a code of 4000 is produced. This results in a dump if SYSABEND or SYSUDUMP cards were 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 was overwritten during the execution of the program.
The most practical solution may be to recompile the program with
SUBSCRIPTRANGE, STRINGSIZE, and STRINGRANGE enabled. Then rerun
the program with the NOSPIE and NOSTAE run-time options. These PL/I conditions
check for possible overwriting by subscripts or substrings that are beyond the
bounds of the referred variable.

If a 4000 ABEND must be run, execute with NOSPIE and NOSTAE.

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 the instruction causing the interrupt. This is done by looking for the
PSW (O1).

3. Inspect this instruction to see if it appears to have been overwritten,
bearing in mind the cause of the interrupt; for example,
   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(O5) 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, see Chapter 8, "Submitting an APAR" on page 115. The cause of
the original entry to the error handler may already be known, 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 not 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 was then made from one of the
addresses in the TCA because of overwriting. Therefore, you should care-
fully check the TCA. (See Appendix A, "Control Blocks" on page 119 for
more information about the TCA.) If this fails to produce results, return to
stage 2 of the above procedure.
Section 3: Locating Specific Information

This section tells you how to find information in a dump. The section is organized in modular form for easy reference. You should look through the list below to discover the items in which you are interested. Suggested methods of debugging a PL/I program from a dump are given in “Section 2: Suggested Debugging Procedures” on page 87. Unless you are experienced in using dumps, or are looking for some particular item, use the procedures in “Section 2: Suggested Debugging Procedures,” rather than attempting to find various items through the information in this section.

Note: In CMS, storage is not divided into subpools. However, the information is still in the dump.

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
P6 Timestamp

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 Find the program interrupt element (PIE) or extended program interrupt element (EPIE)

Stand-Alone Dumps

S1 Finding key areas in stand-alone dumps

Housekeeping Information in All Dumps

H1 Following the DSA back-chain
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 ISA free-area chain
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 requested 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 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.
2. If the interrupt was a program check interrupt (see Figure 34 on page 95), 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 34 on page 95.

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 interrupts, 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 0 for program check interrupts, and to 1 for other interrupts.

Finding Register Values If Interrupt Occurred in Library Routine: If the 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 back-chain 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 DSA, as follows:

<table>
<thead>
<tr>
<th>Bit 4</th>
<th>Bit 5</th>
<th>DSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Procedure block</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Begin block</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>ON-unit</td>
</tr>
</tbody>
</table>

Figure 33. DSA Flag Bits 4 and 5
Figure 34. The Contents of IBMBERR's DSA After a System Detected and a PL/I Interrupt
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 in 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 back-chain 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 is always headed by a flag word of hexadecimal X'8800EEEE', 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 35 on page 97 for details of how to follow the DSA chain.
### Figure 35. The Chaining of DSAs

#### 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. If NOTRACE is specified, the TCA is in subpool 1, preceded by the characters ZTCA.

#### P6: Timestamp

If the TSTAMP installation option is specified in your installation, the date and time of compilation are in the last 16 bytes of the static control section. The first word gives the offset to the information. The static control section is addressed by register 3. If the BLOCK option is specified, the timestamp is printed at the head of the static blocks.
Key Areas of an ABEND Dump

O1: 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.

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 in the link-edit map can then be subtracted from this address; the offset compared to the listing gives the appropriate statement number.

Finding the statement number is not likely to prove useful because of the circumstances in which a 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 4000, see “O6: Finding the Program Interrupt Element (PIE/EPIE).”

O2: Type of Interrupt
The type of interrupt can be found from the first word of the PSW.

O3: Register Contents at the Point of Interrupt
Registers 14 through 2 appear in the PIE (program interrupt element). Registers 3 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 36 on page 103.

O5: The TCA
Register 12 should point at the TCA.

O6: Finding the Program Interrupt Element (PIE/EPIE)
The program interrupt element (PIE) or extended program interrupt element (EPIE) is found in subpool 5. The PIE/EPIE is followed by registers 3 through 13 and then the STAE/ESTAE work area. The STAE/ESTAE 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.

Housekeeping Information in All Dumps

H1: Following the DSA Back-chain
Each DSA holds a back-chain 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 back-chain 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 back-chain 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' back-chain 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.

(For the 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, the programmer 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 '1B'.) 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 under "Using the Table of Offsets" on page 101. 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. (See Figure 33 on page 94.) 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). For more information, see Figure 33 on page 94.

b. The register 14 address saved in the DSA (offset 12 X'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 step 1 on page 99 or step 2 on page 99. 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 '11' 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; 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'7FF', 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'8FF', 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'18FF', the binary would be '0001 1000 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 third 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 for which you are searching. Following this offset, the statement number is given in 2-byte hexadecimal format.
Procedure Name: To find the entry point name, a back-chain 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 DSAs have flag bits 4 and 5 set to '001B. 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 of 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 that, 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 is for IBMBERR. The first byte of the error code is held in this DSA at offset 84 (X'54'). Compare this byte with the values in Figure 32 on page 86. This error code is given an associated PL/I condition. It is the ON-unit for this condition that is 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. If this is possible, chain back one more DSA and try again. (To check if this has occurred, see step 3 on page 100.)

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: Associating Instruction with Correct Statement and Program Block" on page 99), 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, described in "H2: Associating Instruction with Correct Statement and Program Block" on page 99.
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 and 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 Figure 34 on page 95. See H4 for associating DSAs with correct code. IBMBERR's DSA can be identified by X'EEE' in bytes 2 and 3.

H8: Finding and Interpreting Register Save Areas

Register save areas are held at offset X'1C'(12) in all DSAs, including DSAs in LWS. Offsets and registers are shown in Figure 36 on page 103. Each DSA holds the register values as they were on exit from its block.

H9: Register Usage

A summary of register usage, showing which registers are always used for a particular purpose, is given in Figure 11 on page 24.

H10: Following the ISA Free-Area Chain

The ISA 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. The chain starts at offset X'1C' (28) 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.
Figure 36. The Register Save Area in the DSA

H11: Finding the Task Variable
The task variable is held in the TCA at offset X'24'(38).
**H12: Block Structure of Program (Static Back-chain)**

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 back-chain.

**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 '00' 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: Associating DSA with Block” on page 101) 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 Program Logic Manual.

**H16: Finding Main Procedure DSA**

The main procedure DSA can be found by following the back-chain 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'2A'(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 and the character value ZDSA before it.

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 IBMBERR are slightly different from normal if the interrupt was a hardware interrupt. See Figure 34 on page 95 for a diagram of IBMBERR’s DSA.
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 sibling). 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 sibling). If there is no younger sibling, 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 37.)

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 37.)

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 37.)

To Find Sibling Tasks: Previously attached sibling tasks (elder sibling) can be found by following the chain held at offset X'24'(36) in the tasking appendage.

Subsequently, attached sibling tasks (younger siblings) 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 sibling 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. (See also Figure 37 on page 106.)
Subtask chain points to most recently attached subtask.
Younger sibling chain
(i.e. tasks with the same attaching tasks that were attached later)
Elder sibling chain
(i.e. tasks with the same attaching task that were attached earlier)

Note: Because tasks are chained in both directions, all relationships be quickly found.
Following the 'younger sibling chain' leads to the attaching task. When the attaching task is reached, the offset that should be the offset to the younger sibling is to the stopper. Thus it is known that the attaching task has been reached.

Figure 37. The Chaining of Tasks Through Their Tasking Appendages
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.

H20: Following the heap free-area chain
The heap free-area chain connects the areas of heap storage that are available
to satisfy ALLOCATE requests.

The chain starts at offset X'78'(120) 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.

H21: Following the heap storage chain
The heap storage chain connects all areas obtained by GETMAIN macro
instructions for use as heap storage. The chain starts at offset X'74'(116) 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.

Finding Variables
The value of the variables in the program at the point of interrupt can be dis-
covered 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 the DSA associated with a block, see "H4: Associating DSA with
Block" on page 101.)

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 the DSA associated with a block, see
"H4: Associating DSA with Block" on page 101.)

V3: Controlled Variables
Controlled variables are addressed by an anchor word that is held in the
pseudo-register vector. This anchor word can be identified from compiled code,
while the PRV offset can be found in the dump. The address of the controlled
variable must be obtained from the PRV in the dump because it is not filled-in
until the ALLOCATE statement is executed.

The address in the pseudo-register vector is the address of the data or, in
certain circumstances, of a descriptor or a locator/descriptor. These fields are
described in Appendix A, "Control Blocks" on page 119. 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 0 088         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 0 088         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 are available in Appendix A, "Control Blocks" on page 119.

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

<table>
<thead>
<tr>
<th>Control Field</th>
<th>Identification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic Variables</td>
<td>See &quot;Variables&quot;</td>
</tr>
<tr>
<td>Back-chain</td>
<td></td>
</tr>
<tr>
<td>DSA back-chain</td>
<td>offset X'4' in DSA</td>
</tr>
<tr>
<td>ONCA back-chain</td>
<td>offset X'0' in ONCA</td>
</tr>
<tr>
<td>BOS</td>
<td>Offset X'8' from TCA</td>
</tr>
<tr>
<td>Beginning of segment</td>
<td></td>
</tr>
<tr>
<td>Controlled variables</td>
<td>see &quot;Variables&quot;</td>
</tr>
<tr>
<td>DCLCB</td>
<td>Deduced from object program listing</td>
</tr>
<tr>
<td>Declare control block</td>
<td></td>
</tr>
<tr>
<td>DCB</td>
<td>addressed from offset X'14'(20) in FCB</td>
</tr>
<tr>
<td>ENVB</td>
<td>offset X'C'(12) in DCLCB</td>
</tr>
<tr>
<td>Environment block</td>
<td>deduced from object program listing</td>
</tr>
<tr>
<td>DED</td>
<td></td>
</tr>
<tr>
<td>Data element descriptor</td>
<td></td>
</tr>
<tr>
<td>Diagnostic statement table</td>
<td>addressed from offset X'8' from entry point of main procedure</td>
</tr>
<tr>
<td>DFB</td>
<td>addressed from offset X'40'(64) in TCA</td>
</tr>
<tr>
<td>Diagnostic file block</td>
<td></td>
</tr>
<tr>
<td>DSA</td>
<td>addressed by register 13 (see P3 and D3)</td>
</tr>
<tr>
<td>Dynamic storage area</td>
<td></td>
</tr>
<tr>
<td>EOS</td>
<td>offset X'C'(12) in TCA</td>
</tr>
<tr>
<td>End of segment</td>
<td></td>
</tr>
<tr>
<td>Event variable</td>
<td>deduced from object program listing and knowledge of parameter lists of I/O and wait modules</td>
</tr>
<tr>
<td>FCB</td>
<td>identified in PL/I dumps. Addressed via PRV and DCLCB</td>
</tr>
<tr>
<td>File control block</td>
<td></td>
</tr>
<tr>
<td>Flow statement table</td>
<td>addressed from offset X'4C'(76) in TCA</td>
</tr>
<tr>
<td>Filename</td>
<td>addressed from offset X'10'(18) in FCB</td>
</tr>
<tr>
<td>ISA Free-area chain</td>
<td>offset X'1C'(28) in implementation-defined appendage, which is addressed from offset X'28'(40) in TCA</td>
</tr>
<tr>
<td>Heap free-area chain</td>
<td>offset X'78'(120) in the implementation-defined appendage</td>
</tr>
<tr>
<td>Heap storage chain</td>
<td>offset X'74'(116) in the implementation-defined appendage</td>
</tr>
<tr>
<td>Locator/descriptor</td>
<td>deduced from object program listing</td>
</tr>
<tr>
<td>LWS</td>
<td>addressed from offset X'48'(72) in every DSA</td>
</tr>
<tr>
<td>Library workspace</td>
<td></td>
</tr>
<tr>
<td>NAB</td>
<td>offset X'4C'(76) in DSA</td>
</tr>
<tr>
<td>Next available byte</td>
<td></td>
</tr>
</tbody>
</table>

Figure 38 (Part 1 of 2). Quick Guide to Identifying Control Fields
### Control Field Identification

<table>
<thead>
<tr>
<th>Control Field</th>
<th>Identification</th>
</tr>
</thead>
<tbody>
<tr>
<td>ONCA</td>
<td>the offset of the associated ONCA is held in a halfword at offset X'2' in each section of LWS</td>
</tr>
<tr>
<td>ON-communications area</td>
<td></td>
</tr>
<tr>
<td>ONCB</td>
<td>offset X'60' (96) in DSA</td>
</tr>
<tr>
<td>ON-control block start of dynamic</td>
<td>offset X'5C' (92) in DSA</td>
</tr>
<tr>
<td>ONCB chain</td>
<td></td>
</tr>
<tr>
<td>first static ONCB</td>
<td>addressed from offset X'70' (112) in DSA</td>
</tr>
<tr>
<td>ON-cells</td>
<td>object program listing and static storage map</td>
</tr>
<tr>
<td>Parameter lists</td>
<td>See P3 and O3</td>
</tr>
<tr>
<td>Register values</td>
<td>Static listing</td>
</tr>
<tr>
<td>Symbol table</td>
<td>Static listing</td>
</tr>
<tr>
<td>Symbol table vector</td>
<td>Static listing</td>
</tr>
<tr>
<td>Statement number table</td>
<td>see diagnostic statement table</td>
</tr>
<tr>
<td>Static storage</td>
<td>addressed by register 3 in compiled code. See P3 and O3.</td>
</tr>
<tr>
<td>Segment number</td>
<td>first two bytes of BOS, or NAB. '00' = 1, 'FF' = 2, etc.¹</td>
</tr>
<tr>
<td>Tasking appendage</td>
<td>addressed from X'2C' (44) in the TCA.</td>
</tr>
<tr>
<td>Task variable</td>
<td>addressed from X'24' (36) in the TCA.</td>
</tr>
<tr>
<td>TCA</td>
<td>addressed by register 12. See P3 and O3.</td>
</tr>
<tr>
<td>Task communications area</td>
<td></td>
</tr>
<tr>
<td>Variables</td>
<td></td>
</tr>
<tr>
<td>automatic</td>
<td>offset from DSA of block in which they are declared. As shown in variables offset map. See V1.</td>
</tr>
<tr>
<td>based</td>
<td>address of the pointer must be deduced from the object program listing. This gives the address of the variable. See V2.</td>
</tr>
<tr>
<td>controlled</td>
<td>PRV offset referenced in compiled code holds latest allocation of the variable. A back-chain through the previous allocation can be made using the header chain. See V3.</td>
</tr>
<tr>
<td>static</td>
<td>offset from register 3 is shown in variable offset map. See V4.</td>
</tr>
<tr>
<td>area</td>
<td>as for other variables depending on storage class. See V5.</td>
</tr>
<tr>
<td>Variables in areas</td>
<td></td>
</tr>
<tr>
<td></td>
<td>find address of area. Find variable from offset within areas shown in compiled code. See V6.</td>
</tr>
</tbody>
</table>

Figure 38 (Part 2 of 2). Quick Guide to Identifying Control Fields

Note: ¹ Except when the first two bytes of NAB are filled with zeros, the first two bytes of BOS are always less than the first two bytes of NAB when a segment needs to be freed.
Section 4: 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.

The DSA chaining differs slightly when a program is multitasking. The back-chain 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 back-chain.

Compiled code and the static control sections generated by the compiler are always held in storage associated with the control task.

Section 5: Special Considerations for CICS

The PL/I-CICS run-time environment is different from either the PL/I batch environment or the PL/I multitasking environment. This is because system services such as program management, storage management, and error handling are requested through the EXEC CICS interface.

As a result of these differences, when you prepare a PL/I program for execution under CICS, you must take special action during the compile and the link edit steps. You can find information about these actions in the OS PL/I Version 2 Programming Guide.

The CICS environment user exit is IBMFXITA. This user exit, if it determines that an ABEND is required, supplies a four-character EBCDIC ABEND code. The User exit is discussed in detail in OS PL/I Version 2 Installation and Customization under MVS.

CICS ABEND codes are different from PL/I ABEND codes. CICS ABEND codes are listed in the OS PL/I Version 2 Programming Guide.

Section 6: User Exit Considerations

IBM supplies a user exit you can modify. You can invoke it during initialization and termination. The IBM-supplied name of the user exit is IBMBXITA. (The CICS User exit name is IBMFXITA.)

The user exit can request the program to ABEND, if it is invoked with the initialization or the termination function code.
If you invoke the user exit during initialization, neither the PL/I environment nor PL/I error handling facilities are established yet. But if you invoke the user exit with the termination function code, the PL/I environment is established. Then if the user exit experiences a program check, and you have enabled error handling facilities, the program will give an ABEND 4044.

Section 7: SYSTEM Option Considerations

The SYSTEM compile-time option determines the format of the parameter list to the MAIN procedure. The SYSTEM option does not necessarily determine the system or sub-system on which you are running. If you try to pass a type of the parameter list that is different from the type specified by the SYSTEM option, you will get unpredictable results. For more information see the *OS PL/I Version 2 Programming Guide*.

You can find the SYSTEM specification in a main PL/I procedure in a constants section toward the beginning of the compiled code. For more information about the location and representation of the the SYSTEM option in the constants section see the section, "Retaining the PL/I Environment - PLIMAIN" in the *OS PL/I Version 2 Programming Guide*.
Chapter 7. Using SSF and CSSF Search Arguments

Use SSF and CSSF search arguments to look for IBM-supported fixes that apply to your problem. By formulating your own search arguments, you can find information about a reported problem and use that information to correct your own problem. If no other problems have been reported, you can discover you have a new problem that you should bring to the attention of IBM software support.

When you search, you use the Software Support Facility (SSF) or Customer Software Support Facility (CSSF, a subset of the SSF database that is searched if MVS INFO/ACCESS is installed). Your search checks the SSF or CSSF database to determine if your problem has been reported. These data bases contain data associated with APARs (Authorized Program Analysis Reports), PTFs (Program Temporary Fixes), and PSP (Preventive Service Planning) buckets.

If you find a fix in your search, you can order it through the Support Center, retrieve the fix from Info/Access, or pull the fix from Data Link.

Formulating the Search Argument

You formulate a search argument using keywords, which are a set of numbers, a set of characters, or a combination of both.

If you have access to RETAIN, you can formulate a search argument like this:
p; 5668918 ABEND8C1 IBMBERRA

If you have access to INFO/ACCESS, your search argument is constructed through using panels. In either case, your search compares all words in the search argument to all records in the area selected. (For more information on using INFO/ACCESS see the Information Access User’s Guide.)

Your search arguments should contain standard forms of keywords (explained below) and specific keywords related to the problem. Common words such as “and,” “the,” “register,” etc., do not comprise an effective search argument. Your search argument compares all the records in “file” of the “library” within the “facility” that has been selected. Each equal comparison is called a “hit.”

You must have a meaningful search argument. If you have an overly general search, an excessive number of hits result, most of which have no bearing on the problem. In a similar way, if you have an overly descriptive search argument no hits may be found.

You must include specific facts about a problem to form an effective search argument.
Using Standardized Keywords

To help you in your search, certain keywords in SSF have been standardized by IBM to reduce the number of ways a word can be spelled. For example, in "ONCODExxxx," "xxxx" represents the ON-code number, as in "ONCODE8095". Another example is "MSGIELxxxxl" where "xxxx" represents the message number, as in "MSGIEL02301". Note that each example is a single word.

Your search argument should consider all possibilities. (For example, MSGIEL0230I could be IEL0230I or IEL0230.)
Chapter 8. Submitting an APAR

Proper documentation is essential when submitting an APAR. The documentation you provide to IBM must be in the proper form and detail so that IBM programming service personnel can reproduce the problem at the IBM programming service location. You must supply the source program with the APAR to enable the problem to be reproduced and analyzed. The service personnel will be able to resolve the problem faster if you reduce the source program to the smallest, least complex form that still contains the problem.

If you are submitting the APAR because a previous APAR was returned, supply the additional requested documentation and be sure to indicate the number of the previous APAR.

Materials to Submit

If you are submitting an APAR for the first time, supply the materials, listed below, that apply to your problem. Submitting all the required materials avoids having the APAR returned to you for additional information, leading to a faster resolution of your problem.

The following checklist summarizes the materials that you must submit with an APAR. A complete description of each of the types of material follows the checklist.

<table>
<thead>
<tr>
<th>Materials</th>
<th>When Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Source or Failing Test Case</td>
<td>Always</td>
</tr>
<tr>
<td>JCL</td>
<td>MVS only</td>
</tr>
<tr>
<td>Load Libraries</td>
<td>Run-time problems only</td>
</tr>
<tr>
<td>Input Data Sets</td>
<td>Run-time problems only</td>
</tr>
<tr>
<td>PL/I Compiler</td>
<td>Only when requested</td>
</tr>
<tr>
<td>PL/I Library</td>
<td>Only when requested</td>
</tr>
<tr>
<td>Compiler Listing</td>
<td>Always</td>
</tr>
<tr>
<td>JCL Listing</td>
<td>MVS only</td>
</tr>
<tr>
<td>CMS Terminal Session Log</td>
<td>CMS only</td>
</tr>
<tr>
<td>Linkage Editor Listing</td>
<td>Run-time problems only</td>
</tr>
<tr>
<td>Run-Time Dump</td>
<td>Run-time problems only</td>
</tr>
<tr>
<td>Applied PTFs and Fixes</td>
<td>Always, or specify no fixes applied</td>
</tr>
</tbody>
</table>

Figure 39. Summary of Requirements for APAR Submission

Note: If you supply machine-readable material on a tape reel, describe how the tape was created.
Original Source Information
You must supply source information in one of three forms:

- Your original source
- The machine-readable source
- A small, but re-creatable, failing test case.

**Note:** If you do not supply one of these three, IBM Programming Service will probably return your APAR.

If you send machine-readable source, submit the information on a nonlabelled tape. Along with the tape, send a hard copy listing of how the tape was created. Carefully pack and clearly identify machine-readable information. Make sure the APAR number is on the tape, so that it can be identified if it is separated from the rest of the material submitted with the APAR.

Depending on the options and conditions you have, the machine-readable source is different. These are listed in the following table. Also, the machine-readable source should not have any \%NOPRINT statements, unless they are relevant to the problem.

<table>
<thead>
<tr>
<th>Options and Conditions</th>
<th>Machine-Readable Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOINCLUDE</td>
<td>The source is the data set assigned to SYSIN for the compile step.</td>
</tr>
<tr>
<td>NOMACRO</td>
<td></td>
</tr>
<tr>
<td>INCLUDE</td>
<td>The source is the data set assigned to SYSIN for the compile step and the source statement library or libraries referenced in %INCLUDE statements in the program.</td>
</tr>
<tr>
<td>MACRO</td>
<td></td>
</tr>
<tr>
<td>Preprocessor failure</td>
<td></td>
</tr>
<tr>
<td>INCLUDE</td>
<td>The source is the SYSPUNCH data set produced by the compiler when the MDECK compiler option is specified.</td>
</tr>
<tr>
<td>MACRO</td>
<td></td>
</tr>
</tbody>
</table>

Figure 40. Machine-Readable Sources

Load Libraries Information
If the failure occurs at run-time and the source called one or more previously compiled modules, then in addition to your original source, you must supply the load libraries containing these modules in machine-readable form.

Input Data Sets Information
If the failure occurs at run-time, you must provide enough input data with your APAR to allow the re-creation of the failure.

PL/I Compiler and PL/I Library
You do not need to send these unless you are specifically asked for them. IBM programming service personnel need them only if they cannot recreate your problem using programming service's own library and compiler.
Compiler Listing

If you think you have a compiler failure, then all listings that you supply must relate to a specific run of the compiler. Do not send information that is derived from separate compilations or runs. These may mislead the programming support personnel.

With your APAR, always send the listing which results from the compilation of the original source. Perform the compilation with the following compiler options in effect unless the opposite option is required to show the failure or unless the option masks the failure.

In addition to the above options, you may also need to do the following:

- If your compilation is performed with either the INCLUDE or MACRO compile-time options and you have a preprocessor failure, specify the INSOURCE compile-time option as well.
- If your problem is a run-time problem, you must specify the GOSTMT compile-time option.

JCL Listing

In MVS, you must provide listings of job control statements used to run the program. If you are having problems with a batch job, show any cataloged procedures you are using in expanded form by specifying MSGLEVEL=(1,1) in the JOB statement.

CMS Terminal Session Log

If your failure occurs while compiling or running a program under CMS, supply the full details of the virtual machine environment. The best way to do this is:

1. Immediately before you invoke the compiler to reproduce the problem, issue the following commands:
   
   QUERY SET
   QUERY TERMINAL
   QUERY VIRTUAL
   QUERY SEARCH
   QUERY DISK *
   QUERY FILEDEF
   QUERY LIBRARY
   QUERY INPUT
   QUERY OUTPUT

2. Invoke the compiler using the PLIOPT command, specifying the compiler options required to produce the relevant output, preferably on a line printer, or, alternatively, at a typewriter terminal.

Submit the entire terminal listing, from LOGON to LOGOFF. If a display terminal is used, spool console input/output using the

CP SPOOL CONSOLE START
command to provide the full details of all input entered and of all responses received.

Linkage Editor Listing
If your problem is a run-time failure, specify the XREF linkage editor option. Submit the linkage editor map. This map, produced when the failing program is link edited, is essential for the analysis of the storage dump.

Run-Time Dump
If your problem occurs during the run of a PL/I program, supply a storage dump with your APAR. If at all possible, provide a formatted PL/I dump produced by the PL/I error-handling facilities by including the following statement in an ERROR ON-unit that is entered when the program fails:

CALL PLIDUMP ('TFHB');

If for some reason a formatted PL/I dump cannot be obtained, supply a storage dump obtained by using the system SYSUDUMP or SYSABEND facilities or by using a stand-alone dump program. If you think you have a run-time failure, the listings must relate to the relevant link editing and execution steps.

Note: Do not send information that is derived from separate compilations or runs. These may mislead the programming support personnel.

Applied Fixes
Also supply with your APAR a list of any program temporary fixes (PTFs) and local fixes applied to either the compiler or to the library. If no fixes have been applied, indicate this specifically with your APAR.
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.

The controls blocks and the pages they can be found on are listed below.

- “Area Locator/Descriptor” on page 120.
- “Area Variable Control Block” on page 121.
- “Aggregate Descriptor Descriptor” on page 122.
- “Aggregate Locator” on page 124.
- “Array Descriptor” on page 125.
- “CICS Appendage” on page 128.
- “Controlled Variable Block” on page 130.
- “Data Element Descriptor (DED)” on page 132.
- “FORMAT DEDs (FEDs)” on page 138.
- “Declare Control Block (DCLCB)” on page 140.
- “Dynamic Storage Area (DSA)” on page 142.
- “Entry Data Control Block” on page 146.
- “Environment Block (ENVB)” on page 147.
- “Event Variable Control Block” on page 151.
- “File Control Block (FCB)” on page 152.
- “Fetch Control Block (FECB)” on page 163.
- “Input/Output Control Block (IOCB)” on page 164.
- “Label Data Control Block” on page 170.
- “Library Workspace (LWS)” on page 171.
- “ON Communications Area (ONCA)” on page 172.
- “ON Control Block (ONCB)” on page 175.
- “PLIMAIN” on page 177.
- “PLISTART Parameter List” on page 178.
- “Record Descriptor (RD)” on page 180.
- “String Locator/Descriptor” on page 181.
- “Structure Descriptor” on page 182.
- “Task Communication Area (TCA)” on page 190.
- “Symbol Table (SYMTAB)” on page 183.
- “TCA Implementation Appendage (TIA)” on page 196.
- “TCA Tasking Appendage (TTA)” on page 200.
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 or AUTOMATIC storage.

How Addressed
From an offset from registers 3 or 13 known to compiled code

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A(Area Variable)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Length</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Flag</td>
<td>Not Used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Offset of End Of Extent (OEE)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Offset of Largest Free Element (LFE)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>End of Chain of Free Elements</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Area Variable</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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 $X'1'$ 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.

When Generated
As far as possible during compilation. Adjustable values are filled in during execution.

Where Held
Static internal control section or AUTOMATIC storage.

How Addressed
From an offset from registers 3 or 13 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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level</td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Byte 0, Bit 0: 0 indicates a Structure Element

Flag Bits

1 (located in Byte 0, Bit 1): Not applicable to structure elements
2-4 (located in Byte 3, Bits 0-2): Not applicable to structure elements

Offset: (14 bits): The offset within the aggregate descriptor descriptor to the entry for the containing structure. The offset is held in multiples of four bytes. The first element of the structure (the major structure element) has its offset field set to all '1'B.

Level: Logical level of identifier in structure

Dimension: (5 bits): Real dimensionality of identifier
Base Element

Byte 0, Bit 0: 1 indicates a Base Element

Flag Bits
1 (located in Byte 0, Bit 1): Last element in structure indicator
   0 = Not last element
   1 = Last element
2-4 (located in Byte 3, Bits 0-2)
   If Flag 2 = 1, element is an AREA
   If Flag 3 = 1, element is a BIT string
   If Flag 4 = 1, element is a GRAPHIC string

Alignment: (6 bits): Alignment stringency

<table>
<thead>
<tr>
<th>Bit value</th>
<th>Decimal value</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>0</td>
<td>bit</td>
</tr>
<tr>
<td>000111</td>
<td>7</td>
<td>byte</td>
</tr>
<tr>
<td>001111</td>
<td>15</td>
<td>halfword</td>
</tr>
<tr>
<td>011111</td>
<td>31</td>
<td>fullword</td>
</tr>
<tr>
<td>111111</td>
<td>63</td>
<td>doubleword</td>
</tr>
</tbody>
</table>

Length: Length (in bytes) of data. Length is zero for strings and AREAs, whose length is held in descriptors.

Level: Logical level of identifier in structure

Dimension: (5 bits): Real dimensionality of identifier
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 or AUTOMATIC storage.

How Addressed

From an offset from registers 3 or 13 known to compiled code.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Address of data aggregate</td>
<td>Word 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Address of descriptor</td>
<td>Word 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 or AUTOMATIC storage.

How Addressed
From an offset from registers 3 or 13 known to compiler code, or from an aggregate locator.

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
PL/I Version 1: 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.
### PL/I Version 2

The first word in the array descriptor is the **RVO** (relative virtual origin). This is followed by three words for each dimension of the array, containing the multiplier and high and low bound for each dimension.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RVO (AO-VO)</td>
<td>Multiplier</td>
<td>High bound</td>
<td>Low bound</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Multiplier</td>
<td>High bound</td>
<td>Low bound</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>Multiplier 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>High bound 2</td>
<td>Low bound 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>etc.</td>
</tr>
</tbody>
</table>

**Note:** Two full words containing multiplier and high and low bound are included for each array dimension.

### RVO

**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.
CICS Appendage

Function
Holds information needed during operation under CICS/OS/VS.

When Generated
During program initialization under CICS/OS/VS.

Where Held
In the program management area at the head of the ISA.

How Addressed
From TCIC offset X'124' in TCA.
<table>
<thead>
<tr>
<th>Address</th>
<th>Field Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A(CICS TCA)</td>
</tr>
<tr>
<td>4</td>
<td>A(CICS CSA)</td>
</tr>
<tr>
<td>8</td>
<td>A(IBMFSTVA) or 0</td>
</tr>
<tr>
<td>C</td>
<td>A(Msg Output Bootstrap)</td>
</tr>
<tr>
<td>10</td>
<td>A(Report/Count Bootstrap)</td>
</tr>
<tr>
<td>14</td>
<td>Terminal ID</td>
</tr>
<tr>
<td>18</td>
<td>Transaction ID</td>
</tr>
<tr>
<td>1C</td>
<td>PL/I Mask, CICS Mask, Command Workspace</td>
</tr>
<tr>
<td>20</td>
<td>Temp1</td>
</tr>
<tr>
<td>24</td>
<td>Temp2</td>
</tr>
<tr>
<td>28</td>
<td>Temp3</td>
</tr>
<tr>
<td>2C</td>
<td>A(DFHSAP, PL/I-CICS Nucleus Interface)</td>
</tr>
<tr>
<td>30</td>
<td>A(PL/I to CICS Macro Interface)</td>
</tr>
<tr>
<td>34</td>
<td>A(PL/I Program Exec. Interface Block)</td>
</tr>
<tr>
<td>38</td>
<td>ABEND Code</td>
</tr>
<tr>
<td>3C</td>
<td>Interrupt Code</td>
</tr>
<tr>
<td>40</td>
<td>Return Address</td>
</tr>
<tr>
<td>44</td>
<td>(PL/I Acquired Storage Chain)</td>
</tr>
<tr>
<td>48</td>
<td>A(Buffer), Message/Count/Rep Records</td>
</tr>
<tr>
<td>4C</td>
<td>Used as Exec. Interface DSA (184 bytes)</td>
</tr>
<tr>
<td>104</td>
<td>User's Exec. Interface Block Copy (76 bytes)</td>
</tr>
</tbody>
</table>

**TCTMP**: This area is used as a temporary workspace by PL/I. It is comprised of the TCTP1, TCTP2, and TCTP3 fields.

**TCPSW**: This area holds the Program Status Word (PSW) at the time of an interrupt. The field, TCINT, holds the interrupt code; TCRTN holds the return address.
### 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 at offset X'4' in the TCA.)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PRVOFF</td>
<td>Word 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Length</td>
<td>Word 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Chain back to previous allocation</td>
<td>Word 3 Held in Pseudo-register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Task Invocation Count</td>
<td>Word 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Description</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Field used for descriptor or locator/descriptor in certain circumstances, (see below)

**Data**

**PRVOFF**: Offset within pseudo-register vector associated with the controlled variable.

**Length**: Length of the total allocation including the four 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.

---

© Copyright IBM Corp. 1985, 1987
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 convey description of data elements to library conversion, stream I/O routines, check condition routines, and PLITEST.

When Generated

During compilation.

Where Held

Static internal control section or external symbol table csect.

How Addressed

From an offset from register 3 known to compiled code or from a pointer in the symbol table.

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.

Flag 1: Also known as Code Byte and Look up Byte, define the data type.

<table>
<thead>
<tr>
<th>Hex Value</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'00'</td>
<td>FIXED BINARY</td>
</tr>
<tr>
<td>X'04'</td>
<td>FIXED DECIMAL</td>
</tr>
<tr>
<td>X'08'</td>
<td>FLOAT</td>
</tr>
<tr>
<td>X'0C'</td>
<td>FREE DECIMAL (an internal form)</td>
</tr>
<tr>
<td>X'10'</td>
<td>FIXED PICTURE BINARY</td>
</tr>
<tr>
<td>X'14'</td>
<td>FIXED PICTURE DECIMAL</td>
</tr>
<tr>
<td>X'18'</td>
<td>FLOAT PICTURE BINARY</td>
</tr>
<tr>
<td>X'1C'</td>
<td>FLOAT PICTURE DECIMAL</td>
</tr>
<tr>
<td>X'20'</td>
<td>non-VARYING CHARACTER</td>
</tr>
<tr>
<td>X'24'</td>
<td>non-VARYING BIT</td>
</tr>
<tr>
<td>X'28'</td>
<td>VARYING CHARACTER</td>
</tr>
<tr>
<td>X'2C'</td>
<td>VARYING BIT</td>
</tr>
<tr>
<td>X'30'</td>
<td>CHARACTER PICTURE</td>
</tr>
<tr>
<td>X'40'</td>
<td>BINARY constant</td>
</tr>
<tr>
<td>X'44'</td>
<td>DECIMAL constant</td>
</tr>
<tr>
<td>X'48'</td>
<td>BIT constant</td>
</tr>
<tr>
<td>X'50'</td>
<td>F/E Format</td>
</tr>
<tr>
<td>X'54'</td>
<td>P Format (arithmetic)</td>
</tr>
<tr>
<td>X'58'</td>
<td>A/B/P Format (character)</td>
</tr>
<tr>
<td>X'5C'</td>
<td>C Format</td>
</tr>
<tr>
<td>X'60'</td>
<td>X-Format</td>
</tr>
</tbody>
</table>
Hex Value | Data Type
---|---
X'64' | COL Format
X'68' | SKIP Format
X'6C' | LINE Format
X'70' | PAGE Format
X'80' | LABEL
X'84' | ENTRY
X'88' | AREA
X'8C' | TASK
X'90' | OFFSET
X'94' | POINTER
X'98' | FILE
X'9C' | EVENT
X'A0' | GRAPHIC Fixed
X'A8' | GRAPHIC Varying

Flag 2: completes the definition, if necessary.

Bits 0&1 = 00 | A-format item
          01 | B-format item
          10 | Character picture format item
          11 | GRAPHIC

Bit 2 = 0 | Fixed constant
          1 | Float constant

Bit 3 = 0 | Not extended float
          1 | Extended float

Bit 4 = 0 | F-format/fixed picture
          1 | E-format/float picture

Bit 5 = 0 | Declared binary
          1 | Declared decimal

Bits 4&5 = 11 | Then DED is for character

Bit 6 = 0 | Short precision
          1 | Long precision

Bit 7 = 0 | Real or length specified (A or B format) aligned bit string.
          1 | Complex (also set if E, F, or P in C-format) or no length specified (A or B format) or unaligned bit string.

All bits for which neither value is defined are set to '0'B. :cc 20

DED for STRING Data

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Flag 1</td>
<td>Flag 2</td>
</tr>
</tbody>
</table>
```
DED for FLOAT Data

```
0 1 2 3
0 Flag 1 Flag 2 precision
```

DED for FIXED Data

```
0 1 2 3 4
0 Flag 1 Flag 2 precision scale±128
```

DED for PICTURE STRING Data

```
0 1 2 3 4
0 Flag 1 Flag 2 L1
4 L2 Picture in Internal Form
```

Flag 1: (X'30')

L1: Length of field with insertion characters

L2: Length of field without insertion characters

Internal Code: The internal code for string pictures is as follows:

<table>
<thead>
<tr>
<th>Code</th>
<th>Picture(hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>X'00'</td>
</tr>
<tr>
<td>9</td>
<td>X'04'</td>
</tr>
<tr>
<td>X</td>
<td>X'1C'</td>
</tr>
</tbody>
</table>

DED for PICTURE DECIMAL Arithmetic Data

```
0 1 2 3 4
0 Flag 1 Flag 2 Precision Scale Factor+128
4 Length of Picture Length of Data Flag 3 Flag 4
```

Flag 1: (X'14' or X'1C')
Flag 3: Describes the mantissa subfield.

| Bit 0 = | Always set to zero |
| Bit 1 = | Drifting S in subfield |
| Bit 2 = | Drifting + in subfield |
| Bit 3 = | Drifting − in subfield |
| Bit 4 = | Drifting $ in subfield |
| Bit 5 = | Total suppression in subfield |
| Bit 6 = | * in subfield |
| Bit 7 = | Always set to zero |

Flag 4: Describes the exponent subfield. It has the same format as Flag Byte 3.

Internal codes for pictures

<table>
<thead>
<tr>
<th>Code</th>
<th>Picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>9</td>
</tr>
<tr>
<td>04</td>
<td>Y</td>
</tr>
<tr>
<td>08</td>
<td>Z</td>
</tr>
<tr>
<td>0C</td>
<td>·</td>
</tr>
<tr>
<td>10</td>
<td>E</td>
</tr>
<tr>
<td>14</td>
<td>K</td>
</tr>
<tr>
<td>18</td>
<td>T</td>
</tr>
<tr>
<td>1C</td>
<td>I</td>
</tr>
<tr>
<td>20</td>
<td>R</td>
</tr>
<tr>
<td>24</td>
<td>CR</td>
</tr>
<tr>
<td>28</td>
<td>DB</td>
</tr>
<tr>
<td>2C</td>
<td>B</td>
</tr>
<tr>
<td>30</td>
<td>S (t)</td>
</tr>
<tr>
<td>34</td>
<td>S (d)</td>
</tr>
<tr>
<td>38</td>
<td>S (s)</td>
</tr>
<tr>
<td>3C</td>
<td>+</td>
</tr>
<tr>
<td>40</td>
<td>+ (d)</td>
</tr>
<tr>
<td>44</td>
<td>+ (s)</td>
</tr>
<tr>
<td>48</td>
<td>− (t)</td>
</tr>
<tr>
<td>4C</td>
<td>− (d)</td>
</tr>
<tr>
<td>50</td>
<td>− (s)</td>
</tr>
<tr>
<td>54</td>
<td>$ (t)</td>
</tr>
<tr>
<td>58</td>
<td>$ (d)</td>
</tr>
<tr>
<td>5C</td>
<td>$ (s)</td>
</tr>
<tr>
<td>60</td>
<td>/ (l)</td>
</tr>
<tr>
<td>64</td>
<td>/ (d)</td>
</tr>
<tr>
<td>68</td>
<td>/ (s)</td>
</tr>
<tr>
<td>6C</td>
<td>. (l)</td>
</tr>
<tr>
<td>70</td>
<td>. (d)</td>
</tr>
<tr>
<td>74</td>
<td>. (s)</td>
</tr>
<tr>
<td>78</td>
<td>. (l)</td>
</tr>
<tr>
<td>7C</td>
<td>. (d)</td>
</tr>
<tr>
<td>80</td>
<td>, (s)</td>
</tr>
<tr>
<td>84</td>
<td>V</td>
</tr>
</tbody>
</table>

**Note:** Abbreviations for internal codes:

- (t) = terminal
- (d) = drifting
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**

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, −, 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**

Program control DEDs are used to describe program control constants and program control variables. Program control DEDs may be 2 or 4 bytes in length.

```
+ --- + --- +
| 0   | 1   | 2   |
+ --- + --- +
| Flag 1 | Flag 2 | Further Bytes as Required |
```

**Flag 1** Also known as the look up byte. Defines the data type.

X'80' Label variable, 2-byte DED.
X'84'  Entry variable, 4-byte DED. The byte at offset 2 contains a code indicating the entry type.

  Offset 2  Entry Type
  X'10'  PL/I
  X'7'  FORTRAN
  X'5'  COBOL
  X'16'  OPTIONS(ASM).

X'88'  Area variable, 2-byte DED.
X'8C'  Task variable, 2-byte DED.
X'90'  Offset variable, 2-byte DED.
X'94'  Pointer variable, 2-byte DED.
X'98'  File variable, 2-byte DED.
X'9C'  Event variable, 2-byte DED.
X'B0'  Label constant, 2-byte DED.
X'B4'  Entry constant, 4-byte DED. The byte at offset 2 contains a code indicating the entry type.

  Offset 2  Entry Type
  X'10'  PL/I
  X'7'  FORTRAN
  X'5'  COBOL
  X'16'  OPTIONS(ASM).

X'B8'  File constant, 2-byte DED.
X'BC'  CONDITION condition name, 2-byte DED.

Flag 2: Flag 2 is used only by Flag 1 look up bytes X'B0' and X'B4'.

Bit 0  With Flag 1 look up byte X'B0' indicates a label on a FORMAT statement.

  With Flag 1 look up byte X'B4' indicates whether this entry is fetchable.

  Bits 1-7  Unused.
FORMAT DEDs (FEDs)

For the meaning of the flag bytes, see “Data Element Descriptor (DED)” on page 132.

DED for F and E FORMAT Items (FED)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Flag 1</td>
<td>Flag 2</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Flag byte 1 = X'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 G FORMAT Items (FED)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Flag 1</td>
<td>Flag 2</td>
<td>Length</td>
<td></td>
</tr>
</tbody>
</table>

Flags

Flag 1 = X'A0' For G-format
Flag 2 = X'C0'

Length is optional.

DED for PICTURE FORMAT Arithmetic Items (FED)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Flag 1</td>
<td>Flag 2</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Copy of DED as for arithmetic picture</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Flag 1: (X'54')

W: Total length of the format field
DED for PICTURE FORMAT Character Items (FED)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Flag 1</td>
<td>Flag 2</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Copy of DED as for arithmetic picture</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Flag 1: (X'58')

W: Total length of the format field

DED for C FORMAT Items (FED)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Flag 1</td>
<td>Flag 2</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>FED for real part</td>
<td>FED for imag. part</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Flag 1: (X'5C')

Note: The complex bit (bit 7) in Flag 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)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Flag 1</td>
<td>Flag 2</td>
<td>Parameter</td>
<td></td>
</tr>
</tbody>
</table>

Flag 1: (X'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)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Flag 1</td>
<td>Flag 2</td>
<td>Length</td>
<td></td>
</tr>
</tbody>
</table>

Flag 1: (X'58')

The difference between A, B, and P (character) formats is given by bits 0 and 1 of Flag 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 a separate static control section for external files, or in a static internal control section for internal files.

**How Addressed**
The address is generated by the linkage editor for external files; it is addressed by an offset from register 3 for internal files.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Pseudo-Register Offset</td>
<td>DFCB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Declared Attributes</td>
<td>DCLA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Invalid OPEN Attributes</td>
<td>DOPA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>A(Environment Block)</td>
<td>DENV</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Offset of Graphics Extension</td>
<td>Offset of Filename Length</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Filename Length</td>
<td>Filename (to 31 characters)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Declared and Invalid OPEN Attributes**

<table>
<thead>
<tr>
<th>Byte Number</th>
<th>Hex. Value</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01</td>
<td>STREAM</td>
</tr>
<tr>
<td></td>
<td>02</td>
<td>RECORD</td>
</tr>
<tr>
<td></td>
<td>04</td>
<td>DISPLAY</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>reserved for (STRING)</td>
</tr>
<tr>
<td>Byte Number</td>
<td>Hex. Value</td>
<td>Attributes</td>
</tr>
<tr>
<td>-------------</td>
<td>------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>2</td>
<td>01</td>
<td>SEQUENTIAL</td>
</tr>
<tr>
<td></td>
<td>02</td>
<td>DIRECT</td>
</tr>
<tr>
<td></td>
<td>04</td>
<td>TRANSIENT</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>INPUT</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>OUTPUT</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>UPDATE</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>BACKWARDS</td>
</tr>
<tr>
<td>3</td>
<td>01</td>
<td>BUFFERED</td>
</tr>
<tr>
<td></td>
<td>02</td>
<td>UNBUFFERED</td>
</tr>
<tr>
<td></td>
<td>04</td>
<td>KEYED</td>
</tr>
<tr>
<td></td>
<td>08</td>
<td>EXCLUSIVE</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>PRINT</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Not used</td>
</tr>
</tbody>
</table>
Dynamic Storage Area (DSA)

Function

Holds housekeeping information, automatic variables, and temporaries for each block.

When Generated

During execution. Allocated by prolog 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 Figure 41 on page 143.

How Addressed

From register 13.
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Flag 0</td>
</tr>
<tr>
<td>4</td>
<td>A(Chain Back)</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
</tr>
<tr>
<td>1C</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
</tr>
<tr>
<td>2C</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td></td>
</tr>
<tr>
<td>3C</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td></td>
</tr>
<tr>
<td>4C</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>Block-Enable Bits</td>
</tr>
<tr>
<td></td>
<td>CENA</td>
</tr>
<tr>
<td>58</td>
<td>A(Attaching DSA) in Dummy</td>
</tr>
</tbody>
</table>
Flags

Flag 0 (CFF0)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CDSA Bit 0 = 1</td>
</tr>
<tr>
<td>1</td>
<td>CONC Bit 1 = 1</td>
</tr>
<tr>
<td>2</td>
<td>COCH Bit 2 = 1</td>
</tr>
<tr>
<td>3</td>
<td>CIMP Bit 3</td>
</tr>
<tr>
<td>4 &amp; 5</td>
<td>CBEG Bits 4 &amp; 5</td>
</tr>
<tr>
<td>6</td>
<td>CDUM Bit 6 = 1</td>
</tr>
<tr>
<td>7</td>
<td>CSUB Bit 7 = 1</td>
</tr>
</tbody>
</table>

- LWS DSA
- ON-cells exist
- Dynamic ONCBs allocated
- Reserved for the Checkout Compiler
- Procedure DSA
- Begin DSA
- Library DSA
- ON-unit DSA
- Dummy DSA
- Subtask dummy DSA

Flag 1 (CFF1)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CFCCM Bit 0 = 1</td>
</tr>
<tr>
<td>1</td>
<td>CRNB Bit 1 = 1</td>
</tr>
<tr>
<td>2</td>
<td>CRCE Bit 2 = 1</td>
</tr>
<tr>
<td>3</td>
<td>COVR Bit 3 = 1</td>
</tr>
<tr>
<td>4</td>
<td>CGTO Bit 4 = 1</td>
</tr>
<tr>
<td>5</td>
<td>CSNT Bit 5 = 1</td>
</tr>
<tr>
<td>6</td>
<td>CSYE Bit 6 = 1</td>
</tr>
<tr>
<td>7</td>
<td>CFFB Bit 7 = 1</td>
</tr>
</tbody>
</table>

- Byte CFFC is meaningful
- Restore NAB on GOTO
- Restore current-enable on GOTO
- Callee can use this DSA
- EXIT DSA
- Statement number table exists
- SYSPRINT is enqueued by this block
- Flags in Flags 2 are valid
Flags 2 (CFF2)

C2LD  Bit 0 = 1  Last PL/I DSA
C2ID  Bit 1 = 1  Ignore DSA for SNAP
C2IN  Bit 2 = 1  ILC DSA after interrupt
C2IC  Bit 3 = 1  Invocation Count in this DSA
C2SY  Bit 4 = 1  Symbolic dump for this DSA
C2FL  Bit 5 = 1  There are TSO line numbers
       Bit 6 = 1  CMPAT(V2)--Fullword subscripts
       Bit 7  Reserved

Control Task Flag

CCFC  Bit 0 = 1  Block has active subtasks
       Bits 1-7  Not used

This flag byte is the only one in the DSA used by the control task without synchronizing with the subtask. The subtask must never change it. This prevents interference between CPU's on a multiprocessing machine.
Entry Data Control Block

Function
Holds information that will enable an entry to be branched to and the correct static back-chain to be set up. Is used as an entry variable or when an entry is passed as a parameter.

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

\[
\begin{array}{c|c}
0 & 1 & 2 & 3 & 4 \\
\hline
0 & \text{Address of entry point} & \text{Word 1} \\
4 & \text{Address of Statically Containing DSA at Time of Assignment} & \text{Word 2} \\
\end{array}
\]

Word 1
- Bit 0 = 0 \quad \text{Address of entry}
- Bit 0 = 1 \quad \text{Address of location containing 8-char. EBCDIC name of entry point}

Word 2: Bit 0 is always set to zero.

Address of Statically Containing DSA: This address is set in register 5 when the assignment is made to the variable. It enables variables in other blocks to be accessed. When assignment is made the address of the current statically containing DSA is set. This will be the correct address for the entry. If it were not, the entry itself would not be known.
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 static control section with the DCLCB for external files. In static internal storage for internal files.

How Addressed
From offset X'C' in the DCLCB
<table>
<thead>
<tr>
<th>Column 0</th>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
<th>Column 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFLA</td>
<td>NFLB</td>
<td>NFLC</td>
<td>NFLD</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NFLA</td>
<td>NFLB</td>
<td>NFLC</td>
<td>NFLD</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Not Used</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A(Blocksize) if NBLK or NPAG</td>
<td>NBLK or NPAG</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>A(Record Length) if NREC or NLIN</td>
<td>NREC or NLIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>A(Number of Buffers) if NBUF</td>
<td>NBUF</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>A(KEYLOC Value) if NLOC or NATN</td>
<td>NLOC or NATN</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>A(Attention Variable) if NKYL</td>
<td>NKYL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>A(BUFFOFF Value) if NOFF or NNDX</td>
<td>NOFF or NNDX</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>A(INDEXAREA Size) if NNCP or NADD</td>
<td>NNCP or NADD</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>A(Password String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>72</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>88</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>92</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>124</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>132</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>136</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>140</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>144</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>148</td>
<td>A(Record String Locator)</td>
<td>NPAS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Flags

#### NFLA

<table>
<thead>
<tr>
<th>Flag</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NCON</td>
<td>0</td>
<td>Consecutive</td>
</tr>
<tr>
<td>NIND</td>
<td>1</td>
<td>Indexed</td>
</tr>
<tr>
<td>NRG1</td>
<td>2</td>
<td>Regional (1)</td>
</tr>
<tr>
<td>NRG2</td>
<td>3</td>
<td>Regional (2)</td>
</tr>
<tr>
<td>NRG3</td>
<td>4</td>
<td>Regional (3)</td>
</tr>
<tr>
<td>NTPM</td>
<td>5</td>
<td>TP(M)</td>
</tr>
<tr>
<td>NTPR</td>
<td>6</td>
<td>TP(R)</td>
</tr>
<tr>
<td>NOTH</td>
<td>7</td>
<td>Other organization</td>
</tr>
</tbody>
</table>

#### NFLB

<table>
<thead>
<tr>
<th>Flag</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFIX</td>
<td>0</td>
<td>Fixed</td>
</tr>
<tr>
<td>NVAR</td>
<td>1</td>
<td>10 Variable</td>
</tr>
<tr>
<td>NUND</td>
<td>2</td>
<td>11 Undefined</td>
</tr>
<tr>
<td>NDEC</td>
<td>2</td>
<td>Decimal</td>
</tr>
<tr>
<td>NTRO</td>
<td>2</td>
<td>TRKOFL &gt;</td>
</tr>
<tr>
<td>NBLO</td>
<td>3</td>
<td>Blocked</td>
</tr>
<tr>
<td>NSPA</td>
<td>4</td>
<td>Spanned</td>
</tr>
<tr>
<td>NASA</td>
<td>5</td>
<td>CTLASA</td>
</tr>
<tr>
<td>N360</td>
<td>6</td>
<td>CTL360</td>
</tr>
<tr>
<td>NEGS</td>
<td>7</td>
<td>GRAPHIC</td>
</tr>
</tbody>
</table>

#### NFLC

<table>
<thead>
<tr>
<th>Flag</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NLVE</td>
<td>0</td>
<td>LEAVE</td>
</tr>
<tr>
<td>NRRD</td>
<td>1</td>
<td>REREAD</td>
</tr>
<tr>
<td>NGKY</td>
<td>2</td>
<td>GENKEY</td>
</tr>
<tr>
<td>NCBL</td>
<td>3</td>
<td>COBOL</td>
</tr>
<tr>
<td>NOWR</td>
<td>4</td>
<td>NOWRITE</td>
</tr>
<tr>
<td>NXAR</td>
<td>5</td>
<td>INDEXAREA</td>
</tr>
<tr>
<td>NTOT</td>
<td>6</td>
<td>TOTAL</td>
</tr>
<tr>
<td>NXAS</td>
<td>7</td>
<td>INDEXAREA with no argument</td>
</tr>
</tbody>
</table>

#### NFLD

<table>
<thead>
<tr>
<th>Flag</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NBUU</td>
<td>0</td>
<td>BUFFERS</td>
</tr>
<tr>
<td>NCPF</td>
<td>1</td>
<td>NCP</td>
</tr>
<tr>
<td>NFPS</td>
<td>2</td>
<td>PASSWORD</td>
</tr>
<tr>
<td>NKEF</td>
<td>3</td>
<td>KEYLENGTH</td>
</tr>
<tr>
<td>NKLC</td>
<td>4</td>
<td>KEYLOC</td>
</tr>
<tr>
<td>NVFY</td>
<td>5</td>
<td>VERIFY</td>
</tr>
<tr>
<td>NNOL</td>
<td>6</td>
<td>NOLABEL</td>
</tr>
<tr>
<td>NABF</td>
<td>7</td>
<td>ADDBUF</td>
</tr>
</tbody>
</table>
NFLE

N226  Bit 0 = 1  2260
NLOK  Bit 1 = 1  Lock (2260)
       Bits 2-3  Not used
NSTL  Bit 4 = 1  SCALARVARYING
NUSA  Bit 5 = 1  ANSCII
NBOF  Bit 6 = 1  BUFOFF
NBFL  Bit 7 = 1  BUFOFF(L)

NFLF

NXML  Bit 0 = 1  Index multiple
NX11  Bit 1 = 1  High index 2311
NX14  Bit 2 = 1  High index 2314
NOTM  Bit 3 = 1  No tape mark
NALT  Bit 4 = 1  Alternate tape
NOFT  Bit 5 = 1  OFL tracks
NXTN  Bit 6 = 1  Extent number
       Bit 7  Not used

NFLG

NFFM  Bit 0 = 1  F-format
NVFM  Bit 1 = 1  V-format
NUFM  Bit 2 = 1  U-format
NSP2  Bit 3 = 1  Spanned
NBL2  Bit 4 = 1  Blocked
       Bits 5-7  Not used

NFLH

NVSM  Bit 0 = 1  VSAM
NFBD  Bit 1 = 1  BUFND
NFBI  Bit 2 = 1  BUFNI
NFBS  Bit 3 = 1  BUFSP
NFSI  Bit 4 = 1  SIS
NFSK  Bit 5 = 1  SKIP
NFBW  Bit 6 = 1  BKWD
NFRS  Bit 7 = 1  REUSE
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 the event variable.

How Addressed
As other variables depending on storage class.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Flags 1</td>
<td>Flags 2</td>
<td>Status</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Anchor for ECB chain</td>
<td>EECH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>A(DECB) or A(CCB) for I/O</td>
<td>EAEC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>A(TCA appendage of task for I/O)</td>
<td>ETCA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>A(DCLCB) or A(FCB) for I/O or A(Called Procedure) for Tasking</td>
<td>EUSI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Statement Number</td>
<td>ESND</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Flags

Flags 1 (EFL0)

<table>
<thead>
<tr>
<th>Flag</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECOM</td>
<td>0</td>
<td>Complete</td>
</tr>
<tr>
<td>EACT</td>
<td>1</td>
<td>Active</td>
</tr>
<tr>
<td>EOF</td>
<td>2</td>
<td>I/O EVENT</td>
</tr>
<tr>
<td>DSP</td>
<td>3</td>
<td>DISPLAY EVENT</td>
</tr>
<tr>
<td>EWIP</td>
<td>4</td>
<td>EV has caused entry to an ON-unit</td>
</tr>
<tr>
<td>ESNF</td>
<td>7</td>
<td>ESNF field contains the statement number</td>
</tr>
</tbody>
</table>

Flags 2 (EFL1)

<table>
<thead>
<tr>
<th>Flag</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECH</td>
<td>0</td>
<td>Chain of ECBs exists</td>
</tr>
<tr>
<td>EDUM</td>
<td>1</td>
<td>Dummy EVENT</td>
</tr>
</tbody>
</table>
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'2' in DCLCB. The PRV address is held at offset X'4' in the TCA.

Common Section
The common section is followed by either the RECORD or STREAM sections.
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8</td>
<td>Eyecatcher</td>
<td>0000</td>
</tr>
<tr>
<td>0</td>
<td>Statement Mask</td>
<td>0000 FFST</td>
</tr>
<tr>
<td>8</td>
<td>A(Invalid Statement Module)</td>
<td>0000 FAIS</td>
</tr>
<tr>
<td>C</td>
<td>A(Library Transmitter)</td>
<td>0000 FATM</td>
</tr>
<tr>
<td>10</td>
<td>A(DCLCB)</td>
<td>0000 FADL</td>
</tr>
<tr>
<td>14</td>
<td>A(DCB) or A(ACB)</td>
<td>0000 FAOB or FACB</td>
</tr>
<tr>
<td>18</td>
<td>A(Open File Chain)</td>
<td>0000 FAFO</td>
</tr>
<tr>
<td>1C</td>
<td>A(data management for in-line I/O)</td>
<td>0000 FAIL</td>
</tr>
<tr>
<td>20</td>
<td>Error Bytes</td>
<td>0000 Not Used</td>
</tr>
<tr>
<td>24</td>
<td>FATA</td>
<td>0000 FATB</td>
</tr>
<tr>
<td></td>
<td>FATB</td>
<td>0000 FATC</td>
</tr>
<tr>
<td></td>
<td>FATC</td>
<td>0000 Not Used</td>
</tr>
<tr>
<td>28</td>
<td>FFLA</td>
<td>0000 FFLB</td>
</tr>
<tr>
<td></td>
<td>FFLB</td>
<td>0000 FFLC</td>
</tr>
<tr>
<td></td>
<td>FFLC</td>
<td>0000 FFLD</td>
</tr>
<tr>
<td>2C</td>
<td>FFLF</td>
<td>0000 FFLF</td>
</tr>
<tr>
<td></td>
<td>FFLF</td>
<td>0000 FFLG</td>
</tr>
<tr>
<td></td>
<td>FFLG</td>
<td>0000 FFLH</td>
</tr>
<tr>
<td>30</td>
<td>Blocksize</td>
<td>0000 Keylength</td>
</tr>
<tr>
<td>34</td>
<td>Record length</td>
<td>0000 FRCL</td>
</tr>
<tr>
<td>38</td>
<td>A(First Free IOCB)</td>
<td>0000 FAFR or FREC</td>
</tr>
<tr>
<td></td>
<td>A(Hidden Buffer for QISAM LOCATE)</td>
<td>0000 FTYP</td>
</tr>
<tr>
<td></td>
<td>FLEN</td>
<td>0000 FLEN</td>
</tr>
<tr>
<td>40</td>
<td>Reserved for the Checkout Compiler</td>
<td>0000 FGAS</td>
</tr>
<tr>
<td>44</td>
<td>FBIF</td>
<td>0000 Not Used</td>
</tr>
<tr>
<td>48</td>
<td>Not Used</td>
<td>0000 Not Used</td>
</tr>
</tbody>
</table>
Statement Mask (FFST)

<table>
<thead>
<tr>
<th>Bit number</th>
<th>Statement + options</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>READ SET</td>
</tr>
<tr>
<td>1</td>
<td>READ SET KEYTO</td>
</tr>
<tr>
<td>2</td>
<td>READ SET KEY</td>
</tr>
<tr>
<td>3</td>
<td>READ INTO</td>
</tr>
<tr>
<td>4</td>
<td>READ INTO KEYTO</td>
</tr>
<tr>
<td>5</td>
<td>READ INTO KEY</td>
</tr>
<tr>
<td>6</td>
<td>READ INTO KEY NOLOCK</td>
</tr>
<tr>
<td>7</td>
<td>READ IGNORE</td>
</tr>
<tr>
<td>8</td>
<td>READ INTO EVENT</td>
</tr>
<tr>
<td>9</td>
<td>READ INTO KEYTO EVENT</td>
</tr>
<tr>
<td>10</td>
<td>READ INTO KEY EVENT</td>
</tr>
<tr>
<td>11</td>
<td>READ INTO KEY NOLOCK EVENT</td>
</tr>
<tr>
<td>12</td>
<td>READ IGNORE EVENT</td>
</tr>
<tr>
<td>13</td>
<td>WRITE FROM</td>
</tr>
<tr>
<td>14</td>
<td>WRITE FROM KEYFROM</td>
</tr>
<tr>
<td>15</td>
<td>WRITE FROM EVENT</td>
</tr>
<tr>
<td>16</td>
<td>WRITE FROM KEYFROM EVENT</td>
</tr>
<tr>
<td>17</td>
<td>REWRITE</td>
</tr>
<tr>
<td>18</td>
<td>REWRITE FROM</td>
</tr>
<tr>
<td>19</td>
<td>REWRITE FROM KEY</td>
</tr>
<tr>
<td>20</td>
<td>REWRITE FROM EVENT</td>
</tr>
<tr>
<td>21</td>
<td>REWRITE FROM KEY EVENT</td>
</tr>
<tr>
<td>22</td>
<td>LOCATE SET</td>
</tr>
<tr>
<td>23</td>
<td>LOCATE SET KEYFROM</td>
</tr>
<tr>
<td>24</td>
<td>DELETE</td>
</tr>
<tr>
<td>25</td>
<td>DELETE DEY</td>
</tr>
<tr>
<td>26</td>
<td>DELETE EVENT</td>
</tr>
<tr>
<td>27</td>
<td>DELETE KEY EVENT</td>
</tr>
<tr>
<td>28</td>
<td>UNLOCK KEY</td>
</tr>
<tr>
<td>29</td>
<td>WRITE FROM KEYTO</td>
</tr>
<tr>
<td>30</td>
<td>WRITE FROM KEYTO EVENT</td>
</tr>
<tr>
<td>31-63</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
## Error Bytes

### FER1

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FTIP</td>
<td>X'02': Input transmit (data set)</td>
</tr>
<tr>
<td>FTOP</td>
<td>X'03': Output transmit (data set)</td>
</tr>
<tr>
<td>FTOM</td>
<td>X'1A': OMR read error</td>
</tr>
<tr>
<td>FTIX</td>
<td>X'1C': Input transmit (index set)</td>
</tr>
<tr>
<td>FTOX</td>
<td>X'1D': Output transmit (index set)</td>
</tr>
<tr>
<td>FTIS</td>
<td>X'1E': Input transmit (sequence set)</td>
</tr>
<tr>
<td>FTOS</td>
<td>X'1F': Output transmit (sequence set)</td>
</tr>
</tbody>
</table>

### FER2

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFEF</td>
<td>X'01': End of file</td>
</tr>
<tr>
<td>FRVZ</td>
<td>X'04': Zero length record variable</td>
</tr>
<tr>
<td>FRVS</td>
<td>X'05': Short record variable</td>
</tr>
<tr>
<td>FRVG</td>
<td>X'06': Long record variable</td>
</tr>
<tr>
<td>FKCN</td>
<td>X'07': Key conversion in character string</td>
</tr>
<tr>
<td>FKDP</td>
<td>X'08': Key duplication</td>
</tr>
<tr>
<td>FKSQ</td>
<td>X'09': Key sequence</td>
</tr>
<tr>
<td>FKSP</td>
<td>X'0A': Key specification (null key)</td>
</tr>
<tr>
<td>FKNF</td>
<td>X'0B': Key not found</td>
</tr>
<tr>
<td>FKNS</td>
<td>X'0C': No space for keyed record</td>
</tr>
<tr>
<td>FNIO</td>
<td>X'0D': No IOCBl available</td>
</tr>
<tr>
<td>FEAC</td>
<td>X'0E': Active event</td>
</tr>
<tr>
<td>FEUP</td>
<td>X'0F': No prior read before rewrite</td>
</tr>
<tr>
<td>FENC</td>
<td>X'10': No completed read before rewrite</td>
</tr>
<tr>
<td>FETO</td>
<td>X'11': Permanent output error</td>
</tr>
<tr>
<td>FRR2</td>
<td>X'12': Zero length record read</td>
</tr>
<tr>
<td>FEOL</td>
<td>X'13': Record referenced outside data set</td>
</tr>
<tr>
<td>FEXX</td>
<td>X'14': Unidentified I/O error</td>
</tr>
<tr>
<td>FEIR</td>
<td>X'15': Incomplete read for update</td>
</tr>
<tr>
<td>FKTP</td>
<td>X'16': TP term address specification</td>
</tr>
<tr>
<td>FEXS</td>
<td>X'17': Different FCB same record request</td>
</tr>
<tr>
<td>FKCB</td>
<td>X'18': Key conversion (negative BINARY number)</td>
</tr>
<tr>
<td>FKSF</td>
<td>X'19': Key specification (X'FF' etc)</td>
</tr>
<tr>
<td>FASQ</td>
<td>X'1B': I/O sequence error</td>
</tr>
<tr>
<td>FESY</td>
<td>X'20': Synad error encountered</td>
</tr>
<tr>
<td>FRVX</td>
<td>X'21': Record length &lt; KEYLEN + RKP</td>
</tr>
<tr>
<td>FERH</td>
<td>X'22': Record already held</td>
</tr>
<tr>
<td>FEVN</td>
<td>X'23': Record on non-mounted volume</td>
</tr>
<tr>
<td>FESP</td>
<td>X'24': Data set cannot be extended</td>
</tr>
<tr>
<td>FEVS</td>
<td>X'25': No virtual storage for VSAM</td>
</tr>
<tr>
<td>FKNR</td>
<td>X'26': No keyrange for insertion</td>
</tr>
<tr>
<td>FENP</td>
<td>X'27': No positioning for sequential read</td>
</tr>
</tbody>
</table>
FER2

| FEUN | X'28' | Attempt to reposition failed |
| FEST | X'29' | Statement number for data set exceeded |
| FIEU | X'2A' | Index upgrade error |
| FEMP | X'2B' | Maximum number of index PTRs |
| FEIP | X'2C' | Invalid index PTRs |
| FESW | X'2D' | Invalid sequential write |

**FTYP:** 6th and 7th characters of library transmitter name

**FLEN:** Length of FCB (including DCB)

**FATA**

| FDBG | Bit 0 = 1 | Open SYSPRINT for error message |
| FSYS | Bit 1 = 1 | SYSPRINT |
| FCTR | Bit 2 = 1 | Reserved for Checkout Compiler |
| FSTR | Bit 3 = 1 | String operation |
| FRIO | Bit 6 = 1 | RECORD |
| FSIO | Bit 7 = 1 | STREAM |

**FATB**

| FBAK | Bit 0 = 1 | BACKWARDS |
| FUPD | Bit 1 = 1 | UPDATE |
| FOUT | Bit 2 = 1 | OUTPUT |
| FIP7 | Bit 3 = 1 | INPUT |
| FDIR | Bit 6 = 1 | DIRECT |
| FSEQ | Bit 7 = 1 | SEQUENTIAL |

**FATC**

| FEGS | Bit 1 = 1 | GRAPHIC option of the ENVIRONMENT attribute |
| FAXS | Bit 2 = 1 | Axes |
| FPRT | Bit 3 = 1 | PRINT |
| FXCL | Bit 4 = 1 | EXCLUSIVE |
| FKYD | Bit 5 = 1 | KEYED |
| FUNB | Bit 6 = 1 | UNBUFFERED |
| FBUF | Bit 7 = 1 | BUFFERED |
**FFLA**

- **FFIX**  
  Bit 0 = 1  F-format

- **FVAR**  
  Bit 1 = 1  V-format

- **FUND**  
  Bit 2 = 1  U-format

- **FBLO**  
  Bit 3 = 1  Blocked

- **FSPA**  
  Bit 4 = 1  Spanned

  Bits 5 & 6  Not used

- **FKLC**  
  Bit 7 = 1  Key in record variable KEYLOC

**FFLB**

- **FCON**  
  Bit 0 = 1  CONSECUTIVE

- **FIND**  
  Bit 1 = 1  INDEXED

- **FRG1**  
  Bit 2 = 1  REGIONAL(1)

- **FRG2**  
  Bit 3 = 1  REGIONAL(2)

- **FRG3**  
  Bit 4 = 1  REGIONAL(3)

- **FTMP**  
  Bit 5 = 1  TP(M)

- **FPR**  
  Bit 6 = 1  TP(R)

- **FOTH**  
  Bit 7 = 1  Other organization

**FFLC**

- **FQSM**  
  X'00'  QSAM

- **FBSM**  
  X'04'  BSAM

- **FBSL**  
  X'08'  BSAM (Load)

- **FQTM**  
  X'0C'  QTAM

- **FQIS**  
  X'10'  QISAM

- **FBIS**  
  X'14'  BISAM

- **FBDM**  
  X'18'  BDAM

- **FVSM**  
  X'1C'  VSAM

**FFLD**

- **FPPT**  
  Bit 0 = 1  Paper tape

- **FPRI**  
  Bit 1 = 1  Printer

- **FURD**  
  Bit 2 = 1  Unit record device

- **FTRM**  
  Bit 3 = 1  The foreground terminal

- **FEFL**  
  Bit 4 = 1  ENDFILE module loaded

- **FPHB**  
  Bit 5 = 1  Possible hidden buffer

- **FEML**  
  Bit 6 = 1  Error module loaded

- **FGKY**  
  Bit 7 = 1  Genkey

**FFLE**

- **FFER**  
  Bit 0 = 1  I/O error

- **FERI**  
  Bit 1 = 1  Permanent input error

- **FERO**  
  Bit 2 = 1  Permanent output error

- **FEOF**  
  Bit 3 = 1  End of file
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FHID</td>
<td>Bit 4 = 1</td>
<td>Hidden buffer in use</td>
<td></td>
</tr>
<tr>
<td>FEOD</td>
<td>Bit 5 = 1</td>
<td>Move required</td>
<td></td>
</tr>
<tr>
<td>FFNV</td>
<td>Bit 6 = 1</td>
<td>Non-SCALARVARYING</td>
<td></td>
</tr>
<tr>
<td>FSTK</td>
<td>Bit 7 = 1</td>
<td>Not used</td>
<td></td>
</tr>
</tbody>
</table>
### FFLF

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPRD</td>
<td>0</td>
<td>Previous READ</td>
</tr>
<tr>
<td>FPRS</td>
<td>1</td>
<td>Previous READ SET</td>
</tr>
<tr>
<td>FPLC</td>
<td>2</td>
<td>Previous LOCATE</td>
</tr>
<tr>
<td>FPRW</td>
<td>3</td>
<td>Previous REWRITE</td>
</tr>
<tr>
<td>FPOP</td>
<td>4</td>
<td>Previous OPEN or READ IGNORE</td>
</tr>
<tr>
<td>FCLS</td>
<td>5</td>
<td>Close in progress</td>
</tr>
<tr>
<td>FICL</td>
<td>6</td>
<td>Implicit close</td>
</tr>
<tr>
<td>FRSL</td>
<td>7</td>
<td>Previous OPEN (resume load) or READ IGNORE(0)</td>
</tr>
</tbody>
</table>

### FFLG

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEPCR</td>
<td>0</td>
<td>ENDPAGE</td>
</tr>
<tr>
<td>FEEX</td>
<td>1</td>
<td>End of extent</td>
</tr>
<tr>
<td>FCOP</td>
<td>2</td>
<td>COPY option active</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Not used</td>
</tr>
<tr>
<td></td>
<td>4 &amp; 5</td>
<td>Reserved for the Checkout Compiler</td>
</tr>
<tr>
<td>FVVF</td>
<td>6</td>
<td>Newly opened print file</td>
</tr>
<tr>
<td>FNOC</td>
<td>7</td>
<td>File not to be closed</td>
</tr>
</tbody>
</table>

### FFLH

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILF</td>
<td>0</td>
<td>In-line I/O</td>
</tr>
<tr>
<td>FILL</td>
<td>1</td>
<td>In-line LOCATE</td>
</tr>
<tr>
<td>FHYP</td>
<td>2</td>
<td>Hyphen at the end of the line</td>
</tr>
<tr>
<td>FRGT</td>
<td>3</td>
<td>Retry get after concatenation</td>
</tr>
<tr>
<td>FCLL</td>
<td>4</td>
<td>Current line unfinished</td>
</tr>
<tr>
<td>FSPL</td>
<td>5</td>
<td>Initial call from IBMBSPL or blanks at the end of record</td>
</tr>
<tr>
<td>FBER</td>
<td>5</td>
<td>Blanks at the end of record</td>
</tr>
<tr>
<td>FNBO</td>
<td>6</td>
<td>New buffer wanted</td>
</tr>
<tr>
<td>FGPI</td>
<td>7</td>
<td>GET prompt issued — input</td>
</tr>
</tbody>
</table>

### Built-in Function Byte (FBIF)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSKY</td>
<td>0</td>
<td>Samekey flag</td>
</tr>
<tr>
<td></td>
<td>1-7</td>
<td>Not used</td>
</tr>
</tbody>
</table>
### Record I/O Section

Offsets are from start of the FCB.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4C</td>
<td>A(Last I/OCB Used) or A(Dummy Buffer for LOCATE)</td>
<td>FALU or FCDA</td>
</tr>
<tr>
<td>50</td>
<td>A(first I/OCB to be Checked) (BSAM)</td>
<td>FACK</td>
</tr>
<tr>
<td>54</td>
<td>Static Chain of I/OCBs (BDAM/DISAM/DSAM/VSAM)</td>
<td>FIOC</td>
</tr>
<tr>
<td>58</td>
<td>A(I/OCB for Last Completed Read)</td>
<td>FALR</td>
</tr>
<tr>
<td>5C</td>
<td>FEMT FEFT FRET FAFB</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>A(error module) When Loaded</td>
<td>FERM</td>
</tr>
<tr>
<td>64</td>
<td>FGAM or FFNC FFLV or FFNF KEYLOC-1 VSAM or Decrementing Line Count</td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>Record Count</td>
<td>FCCT</td>
</tr>
<tr>
<td>6C</td>
<td>A(Dummy Key Area)</td>
<td>FAKY</td>
</tr>
<tr>
<td>70</td>
<td>Size of I/OCB (BDAM/BISAM) or Current Relative Block (BSAM)</td>
<td>FIOS or FREL</td>
</tr>
<tr>
<td>74</td>
<td>A(Exclusive Block FILE)</td>
<td>FXBA</td>
</tr>
<tr>
<td>78</td>
<td>Offset Table Used in Record Checking</td>
<td>FRTB</td>
</tr>
<tr>
<td>7C</td>
<td>Base OPTCD for RPL (VSAM)</td>
<td>FOPT</td>
</tr>
<tr>
<td>80</td>
<td>A(FCB) or A(FAFB)</td>
<td>FAWB</td>
</tr>
</tbody>
</table>

**FEMT:** 7th character of the error module name

**FEFT:** 7th character of the endfile module name

**FRET:** Data management return code (regional output)

**FAFB:** Work byte for associated files

**FFNC:** Function byte
FARF Bit 0 = 1 READ file
FAPF Bit 1 = 1 PUNCH file
FAWF Bit 2 = 1 PRINT file
FOMR Bit 3 = 1 OMR (no other lists on)
FRFI Bit 4 = 1 R in FUNC option
FPFI Bit 5 = 1 P in FUNC option
FPWI Bit 6 = 1 W in FUNC option
FASC Bit 7 = 1 Associated file

**FFLV:** VSAM flags

FKSD Bit 0 = 1 KSDS
FESD Bit 1 = 1 ESDS
FRDS Bit 2 = 1 RRDS
FPTH Bit 3 = 1 ALTERNATE INDEX PATH
FNUM Bit 4 = 1 ALTERNATE INDEX PATH (non-unique)
FSKP Bit 5 = 1 SKIP
Bit 6 = 1 Not used
FPLO Bit 7 = 1 Position lost

**FCNF:** Conflict byte

FPII Bit 0 = 1 Prior READ invalid
FPPI Bit 1 = 1 Prior PUNCH invalid
FPWI Bit 2 = 1 Prior PRINT invalid
FPLI Bit 3 = 1 Prior PRINT last line invalid
Bit 4-7 Not used

**Stream I/O Section**

Offsets are from the start of the FCB.
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>4C</td>
<td>A(Next Available Byte in a Buffer)</td>
<td>FCBA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>Bytes Remaining in Buffer</td>
<td>Value of Count Built-in Function</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>Page Size</td>
<td>Line Size</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>Current Line No.</td>
<td>Buffer Size</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5C</td>
<td>A(Copy Position in Buffer) or A(Next TPUT Position) for OUTPUT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>A(DCLCB for COPY file)</td>
<td>FCPF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>A(Copy Module Input or A(Tab Module Output)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>Record Count</td>
<td>FRCT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6C</td>
<td>F(SIOCB)</td>
<td>FSCB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Fetch Control Block (FECB)

Function

The FECB is used to contain information about modules specified in FETCH statements.

How Addressed

FECBs are chained together. The chain starts in field TFEP, which is held in the TIA at offset X'3C.'

Where Held

FECBs are set up by IBMBPFR in non-LIFO storage.

When Generated

When a module is fetched.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Chain Field</td>
<td>ZFCH</td>
</tr>
<tr>
<td>4</td>
<td>PRV Offset</td>
<td>ZFPO</td>
</tr>
<tr>
<td>8</td>
<td>Name of Module (8 bytes)</td>
<td>ZFNM</td>
</tr>
<tr>
<td>10</td>
<td>AMODE Switching Code</td>
<td>ZTRFCDE</td>
</tr>
<tr>
<td>20</td>
<td>A(Fetched module entry point)</td>
<td>ZTARGET</td>
</tr>
<tr>
<td>24</td>
<td>A(Call R14 Save Area)</td>
<td>ZSAVR14</td>
</tr>
</tbody>
</table>
### 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 OPEN.

#### Where Held

In non-LIFO storage for VSAM, in subpool 0 for BSAM (obtained by GETPOOL), BISAM or BDAM (obtained in subpool 1 for non-multitasking, 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

<table>
<thead>
<tr>
<th>Field No.</th>
<th>Description</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Static Forward Chain</td>
<td>ICHN</td>
</tr>
<tr>
<td>4</td>
<td>Chain of Free or Unchecked IOCBs or Region Number, Left Adjusted (BDAM)</td>
<td>INXT or IRGN</td>
</tr>
<tr>
<td>8</td>
<td>IFLA IFLB Error Codes (IERR)</td>
<td>IRCB</td>
</tr>
<tr>
<td>10</td>
<td>1st Word of Record Descriptor; A(RCD)</td>
<td>IORD</td>
</tr>
<tr>
<td>14</td>
<td>2nd Word of Record Descriptor; Flags and Record Length</td>
<td>IORL</td>
</tr>
<tr>
<td>18</td>
<td>1st Word of Key Descriptor</td>
<td>IOKD or IFNA</td>
</tr>
<tr>
<td>1C</td>
<td>2nd Word of Key Descriptor</td>
<td>IOKL or IFBK</td>
</tr>
<tr>
<td>20</td>
<td>A(EVENT Variable)</td>
<td>IEVT</td>
</tr>
</tbody>
</table>
Flag Byte (IFLA)

<table>
<thead>
<tr>
<th>Code</th>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFXV</td>
<td>0 = 1</td>
<td>Record locked</td>
</tr>
<tr>
<td>IFMU</td>
<td>1 = 1</td>
<td>Record to move flag</td>
</tr>
<tr>
<td>IFSU</td>
<td>2 = 1</td>
<td>Varying string with non-scalar variable</td>
</tr>
<tr>
<td>IFUS</td>
<td>3 = 1</td>
<td>IOC B in use</td>
</tr>
<tr>
<td>IFER</td>
<td>4 = 1</td>
<td>General error flag</td>
</tr>
<tr>
<td>IFDR</td>
<td>5 = 1</td>
<td>Dummy records are being printed or displayed</td>
</tr>
<tr>
<td>IFDB</td>
<td>6 = 1</td>
<td>Dummy buffer acquired</td>
</tr>
<tr>
<td>IFCH</td>
<td>7 = 1</td>
<td>IOC B checked</td>
</tr>
</tbody>
</table>

Flag Byte (IFLB): Code byte containing offset within 'look-up' table used for record checking

Error Codes (IERR)

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEOF</td>
<td>X'01'</td>
</tr>
<tr>
<td>ITID</td>
<td>X'02'</td>
</tr>
<tr>
<td>ITO P</td>
<td>X'03'</td>
</tr>
<tr>
<td>IRVZ</td>
<td>X'04'</td>
</tr>
<tr>
<td>IRVS</td>
<td>X'05'</td>
</tr>
<tr>
<td>IRVG</td>
<td>X'06'</td>
</tr>
<tr>
<td>IKCN</td>
<td>X'07'</td>
</tr>
<tr>
<td>IKDP</td>
<td>X'08'</td>
</tr>
<tr>
<td>IKSQ</td>
<td>X'09'</td>
</tr>
<tr>
<td>IKSP</td>
<td>X'0A'</td>
</tr>
<tr>
<td>IKNF</td>
<td>X'0B'</td>
</tr>
<tr>
<td>IKNV</td>
<td>X'0C'</td>
</tr>
<tr>
<td>INIO</td>
<td>X'0D'</td>
</tr>
<tr>
<td>IEAC</td>
<td>X'0E'</td>
</tr>
<tr>
<td>IEUP</td>
<td>X'0F'</td>
</tr>
<tr>
<td>IENC</td>
<td>X'10'</td>
</tr>
<tr>
<td>IETO</td>
<td>X'11'</td>
</tr>
<tr>
<td>IRRZ</td>
<td>X'12'</td>
</tr>
<tr>
<td>IEO</td>
<td>X'13'</td>
</tr>
<tr>
<td>IEXX</td>
<td>X'14'</td>
</tr>
</tbody>
</table>

IOLK: Flags and key length

IREF: Relative block or record number (2 words) (BDAM)

IFNA: Next address feedback (BDAM spanned)

IFBK: BDAM feedback (BDAM spanned)
## Non-VSAM Section

This section starts at offset X'24'.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1</td>
<td>Reserved</td>
</tr>
<tr>
<td>2-3</td>
<td>Reserved</td>
</tr>
<tr>
<td>4</td>
<td>IADE or IXLV or IRLB</td>
</tr>
<tr>
<td>24</td>
<td>A(ECB) for Regional Sequential Only or A(Exclusive Block) for Direct Only or A(Binary Region No.- Regional(1) Update</td>
</tr>
<tr>
<td>28</td>
<td>A(Implementation Appendage)</td>
</tr>
</tbody>
</table>

### Data Management Event Control Block

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1</td>
<td>Reserved</td>
</tr>
<tr>
<td>2-3</td>
<td>Reserved</td>
</tr>
<tr>
<td>4</td>
<td>IECB</td>
</tr>
<tr>
<td>2C</td>
<td>BDAM Exception Codes in 2nd &amp; 3rd Bytes</td>
</tr>
<tr>
<td>30</td>
<td>I/O Operation Type Set by Data Mgmt.</td>
</tr>
<tr>
<td>34</td>
<td>A(Data Control Block)</td>
</tr>
<tr>
<td>38</td>
<td>A(Buffer) or A(Record Variable)</td>
</tr>
<tr>
<td>3C</td>
<td>A(Status Indicators) (BSAM &amp; BDAM) or A(Logical Record)</td>
</tr>
<tr>
<td>40</td>
<td>A(Dummy Buffer) (BSAM) or A(Next Record Feedback) (BSAM) or A(KEY) (BDAM and BISAM)</td>
</tr>
<tr>
<td>44</td>
<td>A(Relative Block or Record) that is, A(IREF) (BDAM) or BISAM Exception Codes</td>
</tr>
<tr>
<td>48</td>
<td>A(Next Record Feedback) or Start of Any Appended Buffer (BSAM)</td>
</tr>
<tr>
<td>4C</td>
<td>Start of Any Appended Buffer (BDAM-or-BISAM)</td>
</tr>
</tbody>
</table>
VSAM Section

This section also starts at offset X'24'.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>A(Dummy Buffer)</td>
<td>IDUB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>A(First Key Area)</td>
<td>IKSV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2C</td>
<td>A(Second Key Area)</td>
<td>IKST</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>Pointer for LOCATE Requests</td>
<td>IPTR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>A(ONKEY)</td>
<td>IONK</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data Management Event Control Block
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>A(Data Management Event Control Blocks)</td>
<td>IEVC</td>
</tr>
<tr>
<td>3C</td>
<td>A(Request Parameter List)</td>
<td>IRPL</td>
</tr>
<tr>
<td></td>
<td>SHOWCB Parameter List</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>A(Header)</td>
<td>ISHO</td>
</tr>
<tr>
<td>44</td>
<td>A(Element)</td>
<td>ISEL</td>
</tr>
<tr>
<td>48</td>
<td>Type Codes</td>
<td>ISTC</td>
</tr>
<tr>
<td>4C</td>
<td>A(Block)</td>
<td>ISBL</td>
</tr>
<tr>
<td>50</td>
<td>A(User Area)</td>
<td>ISAR</td>
</tr>
<tr>
<td>54</td>
<td>Length of User Area</td>
<td>ISLN</td>
</tr>
<tr>
<td>58</td>
<td>Element Codes</td>
<td>ISEC</td>
</tr>
<tr>
<td>5C</td>
<td>User Area MODCB Parameter List</td>
<td>ISUA</td>
</tr>
<tr>
<td>60</td>
<td>A(Header)</td>
<td>IMHD</td>
</tr>
<tr>
<td>64</td>
<td>A(Element) Maximum of 3</td>
<td>IMEL</td>
</tr>
<tr>
<td>68</td>
<td>A(Element)</td>
<td>IMEL</td>
</tr>
<tr>
<td>6C</td>
<td>A(Element)</td>
<td>IMEL</td>
</tr>
<tr>
<td>70</td>
<td>MODDCB Type Codes</td>
<td>IMTC</td>
</tr>
<tr>
<td>74</td>
<td>A(Block)</td>
<td>IMBL</td>
</tr>
<tr>
<td>78</td>
<td>Not Used</td>
<td>IM2C</td>
</tr>
<tr>
<td>7C</td>
<td>Area</td>
<td>IARA</td>
</tr>
<tr>
<td>80</td>
<td>Not Used</td>
<td>IM2D</td>
</tr>
<tr>
<td>84</td>
<td>Area Length</td>
<td>IARL</td>
</tr>
<tr>
<td>88</td>
<td>Not Used</td>
<td>IM30</td>
</tr>
<tr>
<td>8C</td>
<td>Key Length</td>
<td>IKYL</td>
</tr>
<tr>
<td>90</td>
<td>Not Used</td>
<td>IM34</td>
</tr>
<tr>
<td>94</td>
<td>OPT Code</td>
<td>IOPT</td>
</tr>
<tr>
<td>98</td>
<td>Not Used</td>
<td>IM35</td>
</tr>
<tr>
<td>9C</td>
<td>Record Length</td>
<td>IRCL</td>
</tr>
</tbody>
</table>
Element control entries start at offset X'78' and continue to end of IOCB. Each entry occupies 2 words, with keyword type code set in 1st half-word as follows:

IMab=X'00ab'

For VSAM files, the IOCB has an associated appendage, comprising the RPL, a dummy buffer if the file has the BUFFERED attribute, and a key save area if the data set is a VSAM KSDS.
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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A(Label Constant) Assigned to the Label Variable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>A(DSA) at the Time of Assignment of Owning Block</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Word 1: bit 0 = 0 Address of label
        = 1 Text reference

Word 2: bit 0 always = 0

Label Constant

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A(Label)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Value to be loaded into Reg. 2 on GOTO It becomes the new base register.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Library Workspace (LWS)

Function
Space reserved for two preformatted 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.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DSA Flags</td>
</tr>
<tr>
<td>4</td>
<td>The same back-chain and register save areas that are shown in the DSA (See the DSA Control Block)</td>
</tr>
<tr>
<td>50</td>
<td>56 Byte Workspace</td>
</tr>
<tr>
<td>88</td>
<td>DSA Flags</td>
</tr>
<tr>
<td>8C</td>
<td>The same back-chain and register save areas that are shown in the DSA (See the DSA Control Block)</td>
</tr>
<tr>
<td>DB</td>
<td>56 Byte Workspace</td>
</tr>
<tr>
<td>110</td>
<td>Current ONCA (see the ONCA Control Block)</td>
</tr>
</tbody>
</table>

**DSA Flags:** These flags are the same as Flag Byte 0 and Flag Byte 1 in the DSA. For further information on these flag bytes and their contents, see “Flags” on page 144.
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'21' in LWS.

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 no interrupt with the requested condition built-in function value has occurred. There is a chain back through all ONCAs to the dummy ONCA.
### String Locators

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>Chain Back to Previous ONCA</td>
</tr>
<tr>
<td>4-7</td>
<td>ONCODE Flag LFG1 Not Used</td>
</tr>
<tr>
<td>8-11</td>
<td>String Locator for ONFILE</td>
</tr>
<tr>
<td></td>
<td>(8 bytes)</td>
</tr>
<tr>
<td>12-15</td>
<td>String Locator for ONCHAR</td>
</tr>
<tr>
<td></td>
<td>(8 bytes)</td>
</tr>
<tr>
<td>16-19</td>
<td>String Locator for ONSOURCE</td>
</tr>
<tr>
<td></td>
<td>(8 bytes)</td>
</tr>
<tr>
<td>20-23</td>
<td>String Locator for ONKEY</td>
</tr>
<tr>
<td></td>
<td>(8 bytes)</td>
</tr>
<tr>
<td>24-27</td>
<td>String Locator for DATAFIELD</td>
</tr>
<tr>
<td></td>
<td>(8 bytes)</td>
</tr>
<tr>
<td>28-31</td>
<td>String Locator for ONIDENT</td>
</tr>
<tr>
<td></td>
<td>(8 bytes)</td>
</tr>
<tr>
<td>32-35</td>
<td>A(Record I/O EVENT Variable)</td>
</tr>
<tr>
<td>36-39</td>
<td>Pointer for ONATTN</td>
</tr>
<tr>
<td>40-43</td>
<td>ONCOUNT</td>
</tr>
<tr>
<td>44-47</td>
<td>Retry Environment</td>
</tr>
<tr>
<td>48-51</td>
<td>Retry Address for Conversion</td>
</tr>
<tr>
<td>52-55</td>
<td>X'40' X'00000000' Flag LFG3</td>
</tr>
<tr>
<td>56-59</td>
<td>LCTI Retry Codes Not Used</td>
</tr>
</tbody>
</table>

### Flag (LFG1)

- **LFOF** Bit 0 = 1  ONFILE valid
- **LFOC** Bit 1 = 1  ONCHAR/ONSOURCE valid
- **LFID** Bit 2 = 1  ONIDENT valid
- **LFKY** Bit 3 = 1  ONKEY valid
- **LFDF** Bit 4 = 1  DATAFIELD valid
- **LFEV** Bit 5 = 1  Associate EVENT variable
- **LFAT** Bit 6 = 1  ONATTN valid
- **LFCT** Bit 7 = 1  ONCOUNT valid
Flag (LFG3)

<table>
<thead>
<tr>
<th>Flag</th>
<th>Bit 0</th>
<th>Bit 1</th>
<th>Bits 2-7</th>
</tr>
</thead>
<tbody>
<tr>
<td>LFSC</td>
<td>0</td>
<td>1</td>
<td>Onsource or ONCHAR is used in an ON-unit</td>
</tr>
<tr>
<td>LFSS</td>
<td>1</td>
<td>1</td>
<td>Onsource set in ONCA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Not used</td>
</tr>
</tbody>
</table>

**LCT1:** Copy of TCA flag byte 1 (TFB1).

**Retry Address (LRAD):** The offset from the base of the library module involved to the address where a conversion is attempted again if ONSOURCE or ONCHAR is 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 DSA.

When Generated
Static ONCBs are generated during compilation, one for each ON statement. Dynamic ONCBs are generated by the prolog 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

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A(previous dynamic ONCB in block) or zero, if first</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDBBC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Qualifier</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDQU</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Condition Code</td>
<td>Flag (LDFFG)</td>
<td>Not Used</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDFFG</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Target</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDTR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Static ONCB

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Condition Code</td>
<td>Flag (LSFFG)</td>
<td>Not Used</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LSTR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Target</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Qualifier: A(DCLCB) for I/O conditions A(SYMTAB) for CHECK A(CSECT) for CONDITION condition

Flag (LDFG and LSFG)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SYSTEM specified</td>
</tr>
<tr>
<td>1</td>
<td>Null ON-unit</td>
</tr>
<tr>
<td>2</td>
<td>GOTO only ON-unit</td>
</tr>
<tr>
<td>3</td>
<td>Condition established</td>
</tr>
<tr>
<td>4</td>
<td>Not Used</td>
</tr>
<tr>
<td>5</td>
<td>Enabled at block entry</td>
</tr>
<tr>
<td>6</td>
<td>Condition enabled</td>
</tr>
<tr>
<td>7</td>
<td>SNAP specified</td>
</tr>
</tbody>
</table>

Target: Address of ON-unit, or offset in DSA of word containing A(label variable).
### 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.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VCON (Primary Entry Point to Program)</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>Zero</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Dummy PLIMAIN**: A control section in IBMBPIRA and IBMTPIRA holding addresses of error message module. This control section is link-edited if no compiler generated PLIMAIN exists.
PLISTART Parameter List

Function

Used to pass housekeeping information extracted by compiler to PL/I initialization routines.

When Generated

PLISTART is a CSECT generated by the compiler for every external compilation. The parameter list is part of the PLISTART CSECT.

General Format of PLISTART

PLISTART contains the three standard entry points PLISTART, PLICALLA, and PLICALLB. When entry is made, addressability is established register 0 pointed at the parameter list and a branch made to entry point A, B, or C of the initialization routine from PLISTART, PLICALLA, and PLICALLB respectively.

The format of the parameter list for PLISTART is given below.

Addressed by register 0
<table>
<thead>
<tr>
<th>Offset</th>
<th>Block Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A(PLIMAIN)</td>
<td>ZYMA</td>
</tr>
<tr>
<td>4</td>
<td>A(SYSPRINT DCLCB)</td>
<td>zero</td>
</tr>
<tr>
<td>8</td>
<td>A(PLIFLOW)</td>
<td>ZYFL</td>
</tr>
<tr>
<td>C</td>
<td>A(PLITABS)</td>
<td>ZYTB</td>
</tr>
<tr>
<td>10</td>
<td>Total length of PRV</td>
<td>ZYPR</td>
</tr>
<tr>
<td>14</td>
<td>Z'FFFF' (ZYV2)</td>
<td>len(PLISTART plist) (ZYBYTES)</td>
</tr>
<tr>
<td>18</td>
<td>A(remote shared library module list) at run time.</td>
<td>ZYAL</td>
</tr>
<tr>
<td>1C</td>
<td>A(PLICOUNT)</td>
<td>ZYCT</td>
</tr>
<tr>
<td>20</td>
<td>A(PLIXOPT)</td>
<td>ZYX0</td>
</tr>
<tr>
<td>24</td>
<td>A(IBMBOPT)</td>
<td>ZYPO</td>
</tr>
<tr>
<td></td>
<td>(X'80000000' = end of release 2 marker)</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>A(PLIXHD)</td>
<td>ZYHD</td>
</tr>
<tr>
<td>2C</td>
<td>A(IBMBEATA) if INTERRUPT compile-time option used</td>
<td>ZYE</td>
</tr>
<tr>
<td>30</td>
<td>X'80000000' (end of release 3 marker)</td>
<td>ZYLTR3</td>
</tr>
<tr>
<td>34</td>
<td>A(Version 2 signature)</td>
<td>ZYSIG</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A(Record Variable)</td>
<td>VRDA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Flag</td>
<td>Length of Data to Transmit</td>
<td>VRDL</td>
<td></td>
</tr>
</tbody>
</table>

Flag (VRDV): These bits indicate the type of INTO or FROM argument as follows:

VRFF  X'00'  For fixed length strings
VRFA  X'01'  For area variables
VRFV  X'02'  For varying length character strings
VRFB  X'03'  For varying length bit strings

Length (VRDL): This field is the 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.
String Locator/Descriptor

Function
Used to pass the address and the length of strings to other routines. Also for handling strings with adjustable lengths for example, 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.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Byte Address of String</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Allocated Length</td>
<td>F</td>
<td>Not Used</td>
<td>F2</td>
</tr>
</tbody>
</table>

\( F = '0' \) 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. Length is held in a number of graphics for graphic 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 a descriptor, which is followed by the offset field for the next base element. If the base element is not a string, array, or area the descriptor field is omitted.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Element Offset from the Start of the Structure</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Element Descriptor (if Required)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Element Offset from the Start of the Structure</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Element Descriptor (if Required)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For every base element in the structure, an entry is made consisting of an offset field and, if the element requires a descriptor, a descriptor.

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
The symbol table holds the name of an identifier during execution and associates it with the address of that identifier. The identifier may be a variable or a program control constant. The symbol table is used only when you have data directed I/O or the CHECK condition, or specify the TEST(.SYM) compile-time option.

Only base elements have symbol tables. Major and minor structure identifiers do not have symbol tables. Base element symbol table names are usually fully qualified.

When Generated
During compilation, if data-directed I/O, the CHECK condition or the TEST(.SYM) compile-time option is used in the program.

Where Held
Static internal control section for internal names. Separate control section for external names.

How Addressed
By an address constant or by an offset from register 3 for internal data, or by an address generated by the linkage editor for external data.

Long Symbol Table Format
Long symbol tables always start on a word boundary. Blanks pad the end of the table to round the symbol table length to a multiple of 4.

Long symbol tables either have the name length field and name appended to the end of the symbol table or have a fullword address which points to the name length field and name.
Short Symbol Table Format

A short symbol table has only the 2-byte flags, 2-byte name length field and the name field. Short symbol tables always start on a word boundary. Blanks pad the end of the table to round the symbol table length to a multiple of 4.

Flags: The flags in the program control constant symbol table have changed in OS PL/I Version 2. In OS PL/I Version 1, the flags in a program control constant symbol table are always set to '0800'X. The program control constant is a label or entry point and the symbol table format is short.

Below are the Version 2 flags for program control constants.
<table>
<thead>
<tr>
<th>Bit 0, 1 and 2</th>
<th>Hex Type and Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'000'B</td>
<td>Static variable or program control constant</td>
</tr>
<tr>
<td>X'001'B</td>
<td>Based variable. Also see bits 12-13.</td>
</tr>
<tr>
<td>X'010'B</td>
<td>Controlled (not parameter) variable</td>
</tr>
<tr>
<td>X'011'B</td>
<td>Defined variable. Also see bits 12-13.</td>
</tr>
<tr>
<td>X'100'B</td>
<td>Automatic variable</td>
</tr>
<tr>
<td>X'101'B</td>
<td>Parameter (not controlled) variable</td>
</tr>
<tr>
<td>X'110'B</td>
<td>Not set by compiler. Used by PLTEST at run-time to flag PLTEST declared variables which are not based.</td>
</tr>
<tr>
<td>X'111'B</td>
<td>Controlled parameter variable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 3</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'0'B</td>
<td>Internal</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 4</th>
<th>Item may appear in some CHECK list or CHECK all. Always set to '1'B, if item is EXTERNAL. Always set to '1'B, if item is label or entry constant (maintains compatibility with version 1 short symbol table).</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'0'B</td>
<td>Item appears in no CHECK list.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 5</th>
<th>Address field A refers directly to data.</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'0'B</td>
<td>Address field A refers to a locator.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 6</th>
<th>A member of a structure.</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'0'B</td>
<td>Not a member of a structure.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Long symbol table.</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'0'B</td>
<td>Short symbol table.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 8</th>
<th>Address field A addresses code.</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'0'B</td>
<td>Address field A does not addresses code.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit 9</th>
<th>Dynamic check enabled.</th>
</tr>
</thead>
<tbody>
<tr>
<td>X'0'B</td>
<td>Dynamic check not enabled.</td>
</tr>
</tbody>
</table>
Bit 10  X'1'B  Dictionary reference precedes symbol table. Used by Checkout Compiler.
        X'0'B  Always zero for Optimizing Compiler.

Bit 11  X'1'B  Isub defined.
        X'0'B  Isub not defined.

Bit 12 and 13  X'11'B  Optimizing BASED implementation. The symbol table describes both the BASED variable and the based pointer qualification. The address field A, the storage flag bits 0-2, the data/locator flag bit 5 and the level number describe the BASED pointer qualification.
        X'10'B  Optimizing DEFINED implementation. The symbol table describes the DEFINED variable and the base variable on which it is defined.
        X'00'B  Not optimizing implementation.

Note: Either storage bits 0-2 or these bits 12-13 indicate a BASED or DEFINED variable. If bits 0-2 are used, the symbol table describes only the BASED or DEFINED variable.

Bit 14  X'1'B  Formerly short symbol table, now long symbol table.
        X'0'B  Now short symbol table.

Bit 15  X'1'B  Symbol table points to identifier 2-byte name length field followed by identifier name.
        X'0'B  Symbol table has identifier name length and name appended to end of symbol table.

Note: The Optimizing Compiler never sets Bits 8-11.

Dimensionality: Dimensionality is the number of dimensions declared for an array item. Dimensionality is zero for other items.

Level Number: The level number is 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 procedure block is 1. The level number is only set for AUTOMATIC, DEFINED, BASED, CONTROLLED parameter, non-CONTROLLED parameter and program control constant. For all others, the number is zero.

When the address field A refers to a DSA offset, the level number accesses the correct DSA. This DSA is either the current DSA or an enclosing block DSA. If you have a GET/PUT DATA or CHECK you may need to use the level number,
since the GET/PUT DATA statement or CHECK condition may occur in a different block than that block in which the variables are declared.

For DEFINED and BASED variables, the level number may be the block of the defined base variable or the based pointer qualification rather than the DEFINED or BASED variable.

The level of the "hypothetical" outer outer block surrounding the external procedure is zero.

Address Fields: Addresses for different data types are held in different formats. As far as possible, addresses are held in address field A. However, sometimes more information is required than can be held in a fullword field. When this happens, address field B is used. See the logic description above for how address fields A and B are used by program control constants and variables.

For data types not listed below the address fields are set to zeros.

Address Field A

If STATIC
the address is the address of data or locator for items that have locators.

If AUTOMATIC
the address is the offset within the DSA of the data or offset of the locator for those items that have locators. The correct DSA is indicated by the block level number.

If CONTROLLED
Bytes 0-1 are zeros. Bytes 2-3 have the offset of the PRV. Flag bit 5 is initialized if the controlled variable has a locator or not.

If BASED
the address describes the declared pointer qualifier, not the based variable. The declared pointer qualifier may be automatic, static or non-parameter controlled. If the declared pointer qualifier information is not present, then:

• Field A is set to zeros
• Storage flag bits 0-2 are set to '001'B
• Data/locator flag bit 5 is set to '1'B
• Bits 12-13 are set to '00'B
• The level number refers to the BASED variable.

If PARAMETER
the address is the offset of the one-word field in the DSA containing the address of corresponding argument within the argument list.

The correct DSA is indicated by the block level number. Flag bit 5 indicates whether the argument has a locator or refers directly to the data.

If CONTROLLED parameter
the address is the offset of the one-word field in the DSA containing the PRV offset for the CONTROLLED variable. The correct DSA is indicated by the block level number. Flag bit 5 indicates whether the variable has a locator or refers directly to the data.

If DEFINED
the address is only supported when base variable is static, automatic or parameter.
For DEFINED variables with locators or descriptors (except when the base variable is STATIC EXTERNAL), the address is the offset within the DSA of the locator or locator/descriptor. Flags are marked AUTO with Bit 5 = '0'B.

For DEFINED variables without locators or descriptors, if the base variable is automatic or static, then this address field A describes the base variable, not the DEFINED variable.

If the base variable is a parameter, the address field A has the offset within the DSA to a one-word field with the address of the data. Flags are marked AUTO with Bit 5 = '0'B.

For DEFINED variables with locators or descriptors and defined upon a base STATIC EXTERNAL variable, this address field A describes the STATIC EXTERNAL base variable.

The correct DSA is indicated by the block level number. When locators or descriptors are built for the DEFINED variable, these items are built within the DSA of the same block in which the DEFINED variable is declared.

If CONDITION condition constant
the address of the 1-byte length field is followed by the condition name truncated to 7 characters. This is the same address which passes to the error handler if a SIGNAL CONDITION(...) statement executes within the program.

If file constant
the address is the DCLCB for the file.

If entry constant
the address is the address of the entry point, if resolved. For FETCHable entries, the address is the PRV offset for this fetchable entry point.

If label constant
the address is address of the associated label constant.

Address Field B

If STRUCTURE (not BASED structure)
the address is the offset from the start of the structure descriptor to the fullword field. This fullword field holds the offset of the base element from the start of the structure. Also see “Structure Descriptor.”

If BASED STRING, BASED AREA or BASED ARRAY
the address is the address of the descriptor within internal static.

If BASED STRUCTURE
the address is the address of the fullword field within the structure descriptor situated in internal static. This holds the offset of the base element from the start of the structure.

Length: The length field contains the number of bytes in the following name field.

Name: The name field contains the fully or partially qualified name.

In PL/I Version 1 symbol tables, the fully qualified names must be <= 256. If the fully qualified name exceeds 256, message IEL0921 is issued and the name is truncated by eliminating names at the highest structure levels, until a partially qualified name is formed with length <= 256.
In OS PL/I Version 2 GET/PUT DATA or CHECK symbol tables, the symbol table name length must be \( \leq 256 \) bytes. In OS PL/I Version 2 PLTEST symbol tables, the symbol table name may exceed 256 bytes and is always fully qualified.
Task Communication Area (TCA)

Function
The TCA is the central communication area for the program. It is used to address the error-handling and storage-management routines, and to point to the current segment of dynamic storage.

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
<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
<th>Symbol(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&quot;Eye Catcher&quot;</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>A(PRV or Zero)</td>
<td>TPRV</td>
</tr>
<tr>
<td>8</td>
<td>Beginning of Segment Pointer (BOS)</td>
<td>TBOS</td>
</tr>
<tr>
<td>C</td>
<td>End of Segment Pointer (EOS)</td>
<td>TEOS</td>
</tr>
<tr>
<td>10</td>
<td>DSA next invocation count</td>
<td>TINC</td>
</tr>
<tr>
<td>14</td>
<td>A(current event variable)</td>
<td>TEVT</td>
</tr>
<tr>
<td>18</td>
<td>A(External Save Area)</td>
<td>TESA</td>
</tr>
<tr>
<td>1C</td>
<td>A(TRT Table for errors)</td>
<td>TTRT</td>
</tr>
<tr>
<td>20</td>
<td>Task Level</td>
<td>TTIC</td>
</tr>
<tr>
<td>24</td>
<td>A(Current Task Variable)</td>
<td>TTSK</td>
</tr>
<tr>
<td>28</td>
<td>A(TCA appendage)</td>
<td>TTIA</td>
</tr>
<tr>
<td>2C</td>
<td>A(Tasking Appendage)</td>
<td>TTTA</td>
</tr>
<tr>
<td>30</td>
<td>A(Save Area for Program Management)</td>
<td>TPSA</td>
</tr>
<tr>
<td>34</td>
<td>Open File Chain Anchor</td>
<td>TOPL</td>
</tr>
<tr>
<td>38</td>
<td>A(Loaded Module List)</td>
<td>TOLD</td>
</tr>
<tr>
<td>3C</td>
<td>Unused</td>
<td>TBUG</td>
</tr>
<tr>
<td>40</td>
<td>A(Diagnostic File Block)</td>
<td>TDFB</td>
</tr>
<tr>
<td>44</td>
<td>PL/I Return Code</td>
<td>TOVV</td>
</tr>
<tr>
<td>4C</td>
<td>A(Flow Statement Number Table)</td>
<td>TSFT</td>
</tr>
<tr>
<td>50</td>
<td>A(Tab Table)</td>
<td>TTAB</td>
</tr>
<tr>
<td>54</td>
<td>A(Flow module)</td>
<td>TEFL</td>
</tr>
<tr>
<td>58</td>
<td>A(LPA Module - Region)</td>
<td>TPSR</td>
</tr>
<tr>
<td>No.</td>
<td>Description</td>
<td>Code</td>
</tr>
<tr>
<td>-----</td>
<td>------------------------------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>0</td>
<td>A(LPA Module - LPA)</td>
<td>TPSL</td>
</tr>
<tr>
<td>60</td>
<td>A(LPA Module - LPA)</td>
<td>TPSM</td>
</tr>
<tr>
<td>64</td>
<td>PRV Initialization Word or Zero</td>
<td>TPRI</td>
</tr>
<tr>
<td>68</td>
<td>A(Module List)</td>
<td>TAML</td>
</tr>
<tr>
<td>6C</td>
<td>A(Get Dynamic Storage Routine)</td>
<td>TGET</td>
</tr>
<tr>
<td>70</td>
<td>A(Free Dynamic Storage Routine)</td>
<td>TFRE</td>
</tr>
<tr>
<td>74</td>
<td>A(Overflow Routine for Get DSA)</td>
<td>TOVF</td>
</tr>
<tr>
<td>78</td>
<td>A(ON Condition Handler)</td>
<td>TERR</td>
</tr>
<tr>
<td>7C</td>
<td>TXAF Not Used</td>
<td>TENV</td>
</tr>
<tr>
<td>80</td>
<td>Normal GOTO Code</td>
<td>TGTC</td>
</tr>
<tr>
<td></td>
<td>Used When GOTO Out of a Block May Occur</td>
<td></td>
</tr>
<tr>
<td>F0</td>
<td>A(EFCL) / Dummy if No FLOW or COUNT</td>
<td>TEFC</td>
</tr>
<tr>
<td>F4</td>
<td>A(Interpretive GOTO Routine)</td>
<td>TGETM</td>
</tr>
<tr>
<td>F8</td>
<td>A(Get Control Routine)</td>
<td>TGCL</td>
</tr>
<tr>
<td>FC</td>
<td>A(Free Control Routine)</td>
<td>TRCL</td>
</tr>
<tr>
<td>100</td>
<td>A(Enqueue SYSPRINT Routine)</td>
<td>TEQR</td>
</tr>
<tr>
<td>104</td>
<td>A(Dequeue SYSPRINT Routine)</td>
<td>TDQR</td>
</tr>
<tr>
<td>108</td>
<td>A(WAIT Routine)</td>
<td>TAHT</td>
</tr>
<tr>
<td>10C</td>
<td>A(COMPLETION Pseudo-variable Routine)</td>
<td>TACP</td>
</tr>
<tr>
<td>110</td>
<td>A(EVENT Assign Routine)</td>
<td>TAEA</td>
</tr>
<tr>
<td>114</td>
<td>A(Priority Routine)</td>
<td>TAPR</td>
</tr>
<tr>
<td>118</td>
<td>A(Enqueue/Dequeue Routines)</td>
<td>TEDR</td>
</tr>
<tr>
<td>11C</td>
<td>Reserved for Users</td>
<td>TUSR</td>
</tr>
<tr>
<td>120</td>
<td>A(Attention Checking Routine)</td>
<td>TATP</td>
</tr>
<tr>
<td>124</td>
<td>A(System Dependant Appendage)</td>
<td>TCIC</td>
</tr>
<tr>
<td></td>
<td>Appendage under CICS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Otherwise not Used)</td>
<td></td>
</tr>
</tbody>
</table>
Flags (TFLG)

Indicate that an abnormal GOTO out of block may take place. Also indicate that certain special error conditions may arise.

Flag Byte 0 (TFB0)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TTIS</td>
</tr>
<tr>
<td>1</td>
<td>TTTT</td>
</tr>
<tr>
<td>2</td>
<td>TTCK</td>
</tr>
<tr>
<td>3</td>
<td>TTFT</td>
</tr>
<tr>
<td>4</td>
<td>ITFD</td>
</tr>
<tr>
<td>5</td>
<td>TTKK</td>
</tr>
<tr>
<td>6</td>
<td>TTDB</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

Note: This flag byte is the only one in the TCA used by the central task without synchronizing with the subtask. The subtask must never change it. This prevents interference between CPUs on a multiprocessing machine.

Flag Byte 1 (TFB1)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TGFD</td>
</tr>
<tr>
<td>1</td>
<td>TGFE</td>
</tr>
<tr>
<td>2</td>
<td>TGFS</td>
</tr>
<tr>
<td>3</td>
<td>TGFS</td>
</tr>
<tr>
<td>4</td>
<td>TGQK</td>
</tr>
<tr>
<td>5</td>
<td>TGTE</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Flag Byte 2 (TFB2)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>THQS</td>
</tr>
<tr>
<td>1</td>
<td>THQI</td>
</tr>
<tr>
<td>2</td>
<td>THCC</td>
</tr>
<tr>
<td>3</td>
<td>THFN</td>
</tr>
<tr>
<td>4</td>
<td>THQF</td>
</tr>
<tr>
<td>5</td>
<td>THQR</td>
</tr>
<tr>
<td>6</td>
<td>THQC</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
Flag Byte 3 (TFB3)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TMDF = 1 Dynamic FLOW set on</td>
</tr>
<tr>
<td>1</td>
<td>TPNR = 1 Prompt not required</td>
</tr>
<tr>
<td>2</td>
<td>Bit 2 = Not used</td>
</tr>
<tr>
<td>3</td>
<td>TNFP = 1 No floating point instructions</td>
</tr>
<tr>
<td>4</td>
<td>TNOF = 1 No FLOW for this GOTO</td>
</tr>
<tr>
<td>5</td>
<td>TISN = 1 Implied SKIP next</td>
</tr>
<tr>
<td>6</td>
<td>TFCT = 1 COUNT required</td>
</tr>
<tr>
<td>7</td>
<td>Bit 7 = Not used</td>
</tr>
</tbody>
</table>

Flag Byte 4 (TXAF)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TX31 = 1 Entry AMODE(31)</td>
</tr>
<tr>
<td>1</td>
<td>TXESPIE = 1 ESPIE in use</td>
</tr>
<tr>
<td>2</td>
<td>TXESTAE = 1 ESTAE in use</td>
</tr>
<tr>
<td>3</td>
<td>TXASYS = 1 Extended architecture</td>
</tr>
<tr>
<td>4-7</td>
<td>Bits 4-7 = Not used</td>
</tr>
</tbody>
</table>

TBOS: The pointer that points the the beginning of the current segment.

TEOS: The pointer that points to the end of the current segment.

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

TTRT: The translate-and-test table contains code used in error handling to identify relevant ON-cells.

TPSA: This points to a preformatted DSA reserved for storage management.

TFOP: Used when closing files at the end of a job.

TORC & TURC: A Standard area to keep these codes.

TOVV: Stack overflow routine for FDAs.

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

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

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

Shared Library (TPSR, TPSL, & TPSM): Used when accessing PL/I library modules in the link-pack-area.

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

Storage Pointers (TGET, TFRE, & TOVF): Entry points to IBMBPGRA that get non-LIFO storage, free non-LIFO storage, and acquire a new segment for LIFO STORAGE.
TERR: Address branched to after a software-detected interrupt occurs.

TENV: Identifies release of libraries being used. See also, "Flag Byte 4 (TXAF)" on page 194.

TRLR: The resident library release number.

TTLR: The transient library release number.

TGTC: 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 under "Handling Flow of Control" on page 32.

TGCL: Routine used in multitasking.

TEQR & TDQR: Library routines used in stream I/O (see Appendix C, "Stream-Oriented Input/Output" on page 241).

TAWT: Address of IBMBJWT, the module used to execute the WAIT statement.

TACP: Address of COMPLETION pseudo-variable module.

TAEA: Address of event assign module.

TAPR: Address of the priority routine.

TEDR: Used for enqueuing and dequeuing files other than SYSPRINT.
TCA Implementation 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.
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
<th>Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8</td>
<td>&quot;Eye Catcher&quot;</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>A(Byte Beyond ISA)</td>
<td>TISA</td>
</tr>
<tr>
<td>4</td>
<td>A(Old PICA)/Fake PICA</td>
<td>TAPC</td>
</tr>
<tr>
<td>8</td>
<td>A(Interrupt Handler)</td>
<td>TERA</td>
</tr>
<tr>
<td>C</td>
<td>Interrupt Mask Flags1 Flags2</td>
<td>TINM</td>
</tr>
<tr>
<td>10</td>
<td>WIT chain anchor</td>
<td>TWTW</td>
</tr>
<tr>
<td>14</td>
<td>Anchor for Chain of Exclusive Blocks</td>
<td>TEXF</td>
</tr>
<tr>
<td>1C</td>
<td>A(Last Free Element)</td>
<td>TLFE</td>
</tr>
<tr>
<td>20</td>
<td>A(Dump Block)</td>
<td>TDUB</td>
</tr>
<tr>
<td>24</td>
<td>A(Dummy DSA)</td>
<td>TDDS</td>
</tr>
<tr>
<td>28</td>
<td>A(Get LWS Routine)</td>
<td>TLWR</td>
</tr>
<tr>
<td>2C</td>
<td>A(Extended Float Simulator)</td>
<td>TASM</td>
</tr>
<tr>
<td>30</td>
<td>Two Words for the Name of the Extended Float Simulator</td>
<td>TSNM</td>
</tr>
<tr>
<td>38</td>
<td>A(Storage Report Information)</td>
<td>TASR</td>
</tr>
<tr>
<td>3C</td>
<td>Chain of Fetched Entry Points</td>
<td>TFEP</td>
</tr>
<tr>
<td>40</td>
<td>A(STAE Exit Routine)</td>
<td>TAST</td>
</tr>
<tr>
<td>44</td>
<td>A(Housekeeping Interrupt Routine)</td>
<td>TERC</td>
</tr>
<tr>
<td>48</td>
<td>A(First Count Table)</td>
<td>TCTF</td>
</tr>
<tr>
<td>4C</td>
<td>A(Last Count Table Used)</td>
<td>TCTL</td>
</tr>
<tr>
<td>50</td>
<td>Saved A(TCA) for the Error Handler</td>
<td>TATC</td>
</tr>
</tbody>
</table>
### Flags1 (TFL1)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Task terminated normally</td>
</tr>
<tr>
<td>1</td>
<td>SYSPRINT open STREAM print</td>
</tr>
<tr>
<td>2</td>
<td>STAE exit in progress</td>
</tr>
<tr>
<td>3</td>
<td>Dump I/O in progress</td>
</tr>
<tr>
<td>4-7</td>
<td>Not used</td>
</tr>
</tbody>
</table>

### Flags2 (TFL2)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Caller provided ISA</td>
</tr>
<tr>
<td>1</td>
<td>Storage report required</td>
</tr>
<tr>
<td>2</td>
<td>STAE required</td>
</tr>
<tr>
<td>3</td>
<td>SPIE required</td>
</tr>
<tr>
<td>4</td>
<td>Syntax error in program management options</td>
</tr>
<tr>
<td>5</td>
<td>Multiple STAE required</td>
</tr>
<tr>
<td>6-7</td>
<td>Not used</td>
</tr>
</tbody>
</table>

### Flags3 (TFL3)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Free heap segments</td>
</tr>
<tr>
<td>1</td>
<td>Heap below the 16M line</td>
</tr>
<tr>
<td>2</td>
<td>Free ISA segments</td>
</tr>
</tbody>
</table>
TXHIT  Bit 3 = 1  Heap initialized
TXHPA  Bit 4 = 1  Heap preallocated
        Bit 5 & 6  Not used
TXNHP  Bit 7  No heap processing

**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.

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

**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

{TWITW}:

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

**TEXF:** Used when handling exclusive files.

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

**TDUB:** Used when a PLIDUMP is being executed.

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

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>POST Event Control Block</td>
</tr>
<tr>
<td>4</td>
<td>Parameter list for Control Task (2 words)</td>
</tr>
<tr>
<td>8</td>
<td>WAIT Event Control Block</td>
</tr>
<tr>
<td>10</td>
<td>A(TCB)</td>
</tr>
<tr>
<td>14</td>
<td>A(ECBLIST Element)</td>
</tr>
<tr>
<td>18</td>
<td>A(TCA)</td>
</tr>
<tr>
<td>1C</td>
<td>Not Used</td>
</tr>
<tr>
<td>20</td>
<td>Chain of Sister Tasking Appendages</td>
</tr>
<tr>
<td>24</td>
<td>Anchor for Subtask Sister Chain</td>
</tr>
<tr>
<td>28</td>
<td>Anchor for I/O EVENT Chain</td>
</tr>
<tr>
<td>2C</td>
<td>A(Attaching DSA)</td>
</tr>
<tr>
<td>30</td>
<td>A(Task Invocation Point)</td>
</tr>
</tbody>
</table>

Post Codes to Control Task
X'0'  Completion pseudo-variable
X'4'  EVENT assignment
X'8'  PRIORITY pseudo-variable
X'C'  I/O EVENT completion
X'10' WAIT termination
X'14' Detach this block
X'18' Dedicate control task
X'1C' Liberate control task
X'20' Attach a task
X'24'  End of task
X'28'  Terminate a subtask
X'2C'  Terminate a subtask
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.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>KFL0</td>
<td>KFL1</td>
<td>Priority</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>A(SYMTAB)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>A(TCA Tasking Appendages)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>A(Calling PROCEDURE)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Flags

KFL0
KACT Bit 0 = 1 Active
Bits 1-7 Not used

KFL1
KDUM Bit 0 = 1 Dummy
KSTE Bit 1 = 1 Symbol table exists
Bits 2-7 Not used
Appendix B. Record-Oriented Input/Output

Introduction

This appendix 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 MVS 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 diagrammatically in Figure 41 on page 204. Figure 55 on page 238 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.
Figure 41. The Principles Used in Record I/O Implementation
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.

If the TOTAL option is used, the majority of transmission statements on buffered consecutive files are compiled as short 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 use to branch directly to the data management routines. When running in an MVS/XA environment, the subroutines set the correct addressing mode (AMODE) for data management. These transmitters are also used for error situations and end-of-file conditions.

The TOTAL option is a method used to inform the compiler that no additional information will be supplied about the file via the DD statement. (That is, that the TOTAL information about the file has been declared.) This allows the compiler to determine whether or not inline I/O statements can be used. The conditions when they are used are described in Figure 56 on page 239 and Figure 57 on page 240.

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 in the file control block (FCB). Any attempt to access the file when it is not open results in control being passed to the open routines in the PL/I libraries. The FCB is mapped in “File Control Block (FCB)” on page 152.
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 42 on page 207 for library subroutines and Figure 44 on page 210 for control blocks. More detailed descriptions for each statement type are given below.
RESIDENT LIBRARY MODULES
- IBMBOCL: Open/Close bootstrap routine
- IBMBRIO: Record I/O interface routine

TRANSIENT LIBRARY MODULES

**Open Modules**
- IBMBOPA: Open error handler
- IBMBOPB: Open routine Phase I
- IBMBOPC: Open routine Phase II
- IBMBOPD: Open routine Phase III
- IBMBOPE: Open routine Phase II (VSAM)
- IBMBOPZ: Direct output file formatter

**Close Module**
- IBMBOCA: Close module

**Transmitter Modules**
- IBMBRAA: Regional sequential output
- IBMBRAB: Regional sequential output
- IBMBRAC: Regional sequential output
- IBMBRAD: Regional sequential output
- IBMBRAE: Regional sequential output
- IBMBRAF: Regional sequential output
- IBMBRAG: Regional sequential output
- IBMBRAH: Regional sequential output
- IBMBRAI: Regional sequential output
- IBMBRAA: Regional sequential input/update
- IBMBRBB: Regional sequential input/update
- IBMBRBC: Regional sequential input/update
- IBMBRBD: Regional sequential input/update
- IBMBRBE: Regional sequential input/update
- IBMBRBF: Regional sequential input/update
- IBMBRGG: Regional sequential input/update
- IBMBRCA: Unbuffered consecutive
- IBMBRBC: Unbuffered consecutive
- IBMBRCC: Unbuffered consecutive
- IBMBRCD: Unbuffered consecutive OMR
- IBMBRCE: Unbuffered consecutive associated file
- IBMBRDA: Regional direct non-exclusive
- IBMBRDB: Regional direct non-exclusive
- IBMBRDC: Regional direct non-exclusive
- IBMBRDD: Regional direct non-exclusive
- IBMBRAA: Indexed sequential input/update
- IBMBRJB: Indexed sequential input/update
- IBMBRKA: Indexed direct non-exclusive
- IBMBRK: Indexed direct non-exclusive
- IBMBRKC: Indexed direct non-exclusive

Figure 42 (Part 1 of 2). Library Subroutines Used in Record I/O
### Transmitter Modules (Continued)

- **IBMBRLA**: Indexed sequential output
- **IBMBRLB**: Indexed sequential output
- **IBMROA**: Buffered consecutive (non-spanned)
- **IBMROB**: Buffered consecutive (non-spanned)
- **IBMROC**: Buffered consecutive (non-spanned)
- **IBMROD**: Buffered consecutive (non-spanned)
- **IBMROE**: Buffered consecutive input (spanned)
- **IBMROF**: Buffered consecutive output (spanned)
- **IBMROG**: Buffered consecutive update (spanned)
- **IBMROH**: Buffered consecutive OMR
- **IBMROI**: Buffered consecutive associated file
- **IBMRTOP**: Teleprocessing file input
- **IBM RVA**: VSAM ESDS transmitter
- **IBMBRVG**: VSAM KSDS sequential output
- **IBMBRVM**: VSAM KSDS other operations and path
- **IBMBRI**: RRDS
- **IBM R CXA**: Exclusive regional direct update update/input
- **IBM R XB**: Exclusive regional direct update update/input
- **IBM R XC**: Exclusive regional direct update update/input
- **IBM R XD**: Exclusive regional direct update update/input
- **IBM R Y A**: Exclusive indexed direct update update/input
- **IBM R Y B**: Exclusive indexed direct update update/input
- **IBM R Y C**: Exclusive indexed direct update update/input
- **IBM R Y D**: Exclusive indexed direct update update/input
- **IBM SOF**: Stream output file
- **IBM SOU**: Stream output file
- **IBM SOV**: Stream output file
- **IBM S T F**: Stream output print file
- **IBM S T I**: Stream input file
- **IBM S T U**: Stream output print file
- **IBM S TV**: Stream output print file
- **IBM S T**: Stream output file
- **IBM S TP**: Stream output file

### Record I/O Error Modules

- **IBMREA**: Record I/O error module
- **IBMREB**: Record I/O error module
- **IBMREC**: Record I/O error module
- **IBMREE**: Record I/O error module
- **IBMREF**: Record endfile module

*Figure 42 (Part 2 of 2). Library Subroutines Used in Record I/O*
### Access Method

The access method used for different PL/I file types is shown in Figure 43.

<table>
<thead>
<tr>
<th>File Type</th>
<th>Access Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffered consecutive</td>
<td>QSAM/VSAM</td>
</tr>
<tr>
<td>Unbuffered consecutive</td>
<td>BSAM/VSAM</td>
</tr>
<tr>
<td>Regional sequential (not spanned records)</td>
<td>BSAM</td>
</tr>
<tr>
<td>Regional sequential (spanned records only)</td>
<td>BDAM</td>
</tr>
<tr>
<td>Regional direct</td>
<td>BDAM</td>
</tr>
<tr>
<td>Indexed sequential</td>
<td>QISAM/VSAM</td>
</tr>
<tr>
<td>Indexed direct</td>
<td>BISAM/VSAM</td>
</tr>
<tr>
<td>TP buffered input/update</td>
<td>TCAM</td>
</tr>
<tr>
<td>VSAM</td>
<td>VSAM</td>
</tr>
</tbody>
</table>

Figure 43. Access Methods and File Types

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.
<table>
<thead>
<tr>
<th>CONTROL BLOCKS GENERATED FROM FILE DECLARATION</th>
<th>CONTROL BLOCK GENERATED DURING EXECUTION OF OPEN STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DCLCB</strong></td>
<td><strong>Open control block (OCB)</strong></td>
</tr>
<tr>
<td>Function: Holds all file attributes used in file declaration</td>
<td>Function: To contain file attributes given in OPEN statement</td>
</tr>
<tr>
<td>Location: Separate control section for external files, static internal for internal files</td>
<td>Location: In static storage</td>
</tr>
<tr>
<td>When generated: During compilation</td>
<td>When generated: During compilation</td>
</tr>
<tr>
<td>Contents:</td>
<td>Contents: The attributes when specified on the OPEN statement</td>
</tr>
<tr>
<td>- Record of file attributes at declaration</td>
<td>- Address of ENVB</td>
</tr>
<tr>
<td>- File name</td>
<td>- Offset of FCB pointer in PRV</td>
</tr>
<tr>
<td>Environment control block (ENVB)</td>
<td></td>
</tr>
<tr>
<td>Function: Holds information on environment options</td>
<td></td>
</tr>
<tr>
<td>Location: In static storage</td>
<td></td>
</tr>
<tr>
<td>When generated: During compilation</td>
<td></td>
</tr>
<tr>
<td>Contents: Addresses of</td>
<td></td>
</tr>
<tr>
<td>- blocksize</td>
<td></td>
</tr>
<tr>
<td>- record length</td>
<td></td>
</tr>
<tr>
<td>- number of buffers</td>
<td></td>
</tr>
<tr>
<td>- KEYLOC value</td>
<td></td>
</tr>
<tr>
<td>- key length</td>
<td></td>
</tr>
<tr>
<td>- indexarea size</td>
<td></td>
</tr>
<tr>
<td>- addbuf</td>
<td></td>
</tr>
</tbody>
</table>

Figure 44 (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 static 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 static 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 transmitters
  - DCB
  - Data Management control block/Access-Method Control Block

Figure 44 (Part 2 of 2). The Fields Used in Implementing Record I/O
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 shown in "Environment Block (ENVB)" on page 147.

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 45 shows the code resulting from a file declaration.

![Figure 45. Information in the File Declaration Is Held in the ENVB and the DCLCB Until the File Is Opened](image-url)
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 explicit 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. File transient open modules are used. Their functions are summarized below.

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 46 on page 215.

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, 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.
1. DCLCB identifies file

2. Open control block (OCB) holds options in OPEN statement

3. Title held in static

OPEN FILE (F2) OUTPUT TITLE ('OUTFILE');

4. Executable instructions call to Open close bootstrap module passing parameter list

5. containing addresses etc for 1, 2 and 3

DCLCB set up during file declaration see figure 8.5

Open control block in static. See Appendix A for Format:

000048 0020000000000800 CONSTANT 00000000

Title (held in static internal) is addressed via locator (also in static internal)

Title

0000A0 D6E4E3C6C9D3C5

Locator

000020 000000A000070000

Machine Instructions

000088 41 10 3 064 LA 1,100(0,3) Point R1 at P-lists
00008C 5B F0 3 00C L 15,A..IBMBOCLA Branch to open/close bootstrap
000090 05 EF BALR 14,15

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 46. OPEN 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 there 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.

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 greater than one. If any errors or discrepancies 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
- SEQ/DIR
- KSDS or PATH
- UPD/NUP
- GEN/FKS

KEQ, MVE, and SYN are always specified.
7. Load the appropriate library transmitter as follows:

```
ESDS IBMBRVAA
KSDS SEQUENTIAL OUTPUT
    IBMBRNGA
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 not 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 47 on page 218.

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 cannot be known until execution. An explanation of the use of the pseudo-register vector is given in Chapter 3, “Compiler Output” on page 11, under the heading “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 47. 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 appendix. 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 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 "Record Descriptor (RD)" on page 180.

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.

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. The implementation of event I/O is covered briefly at the end of this appendix.

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 48 on page 221.
COMPILATION

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 088 LA 9,184(0,13) Pick up address record descriptor
   000096 50 90 3 084 ST 9,132(0,3) Place in parameter list
   000098 41 90 D 080 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, .1MBRIOA
   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 48 (Part 1 of 2). Handling a Transmission Statement
EXECUTION OF TRANSMISSION STATEMENT

Call from compiled code

IBM4890
(Resident library interface module)
Loads parameters into registers.
Calls PL/I transient library transmitter whose address is placed in the FCB during the execution of the OPEN statement.

PL/I TRANSMITTER
(Transient library module
Calls data management.
Checks for errors and moves record and key if necessary

DATA MANAGEMENT
Handle the transfer of data

Return to compiled code

Figure 48 (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 is made to this point to continue the statement. (See "Implicit Open for Library-Call I/O" on page 229, for further discussion of this topic.)

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 appendix 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 "Input/Output Control Block (IOCB)" on page 164. The format of the DECB and a further description of its use is given in the publications *OS/VS2 MVS Data Management Macro Instructions*, or in *MVS/Extended Architecture 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 49 on page 225.
Figure 49. Handling the EVENT Option

Key

- READ EVENT statement
- WAIT statement
- Further PL/I statements
Running

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” on page 229.)

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.

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 42 on page 207. 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 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 for Library-Call I/O” on page 229 for further details.)

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 50 on page 228.
DCLCB identifies file to be closed

CLOSE FILE (F2)

Executable instructions consist of a call to the open/close bootstrap module passing parameter list

DCLCB set up for file declaration see figure 8.5

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 bootstrap
0000B4 05 EF BALR 14,15

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

Call from compiled code

IBMBOCL
Entry point C
Resident library open/close bootstrap routine. Calls the close routine

IBMBOCA
Transient library close routine. Calls transmitter to complete I/O if necessary. Calls data management to close the data set. Removes FCB from Open File Chain. Restores PRV offset to point to dummy FCB.

DATA MANAGEMENT
Disassociates file from data set.

PL/I TRANSMITTER
Transient library routine. Calls data management to complete I/O

Return to compiled code

Figure 50. The Execution of an Explicit CLOSE Statement
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 51 on page 230).

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 attempted again 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 52 on page 231.)
Figure 51. The Addressing Mechanism Used during Implicit Open
Record I/O Error Module | File Types
---|---
IBMBREA | Consecutive buffered
IBMBREB | Indexed
IBMBREC | Regional, consecutive unbuffered, and transient
IBMBREE | VSAM
ENDFILE Module | All SEQUENTIAL/INPUT/UPDATE file types (excluding VSAM)
IBMBREF |

Figure 52. 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 52).

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. IBMBRIOB 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.
Contents FEFT
Initialized by open routine with character "A", "B", "C", "E" indicating general error support module.
Altered by end of file routines in transmitter to character "F" indicating ENDFILE module.

Contents FEMT
Always contains character indicating general error support module.

IBMBRIIO
(entry point B)
Loads and calls module indicated in "FEFT" and places its address in FERM.

IBMBREF
Endfile module
If ENDFILE:
Calls error handler
If other error:
Loads and calls error module indicated in "FEMT". Placing address in FERM.

IBMBRE/A/B/C/E
General error support modules.
Handle all errors including ENDFILE.

Key
- - - - - - If no errors have occurred.
• • • • • • • If 1st. error was ENDFILE and no other errors occurred.
--- --- --- If non-ENDFILE errors have occurred.

Figure 53. The Fields Used in Record I/O Error Handling
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.

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. 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 short in-line calls to the data management routines (see Figure 56 on page 239 and Figure 57 on page 240 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 full library subroutines.

Executable Instructions

For in-line I/O, a call is made to a special entry in a transmitter. In an MVS/XA environment, this transmitter provides the correct addressing mode and directly calls the data management routine via the address held in the FCB for output files, and in the DCB for input files. 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 on output files, compiled code does not call data management direct. Instead a call is made to another short call 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, unblocking 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 54 on page 236.

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 51 on page 230). 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.
**SOURCE STATEMENTS**

1    TOTAL: PROC OPTIONS(MAIN);
2     DCL LINE FILE RECORD INPUT
     ENV(FB,RECSIZE(80),BLKSIZE(400),TOTAL);
3     DCL CARD CHAR(80);
4     READ FILE(LINE) INTO(CARD);
5     END TOTAL;

* STATEMENT NUMBER 4

00005E 18 72  LR 7,2  Save program base
000060  58 F0 3 024  L 15,36(0,3)  Load R15 address of DCLCB
000064  18 BF  LR 11,15  Load R11 DCLCB
000066  58 10 C 004  L 1,4(0,12)  Load R1 PRV
00006A  5A 10 8 000  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  Load address of DCB
000078  BF 17 8 04D  ICM 1,7,77(8)  Get last record address
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 Library call
000088  18 18  LR 1,8  Restore DCB address if a new
                     buffer is required
00008A  41 80 3 028  LA 8,40(0,3)  Pass abnormal return address
                     (CL.3 in R8 for error
                     handling
00008E  58 F0 2 01C  L 15,28(0,2)  Get short transmitter
000092  05 EF  BALR 14,15  Branch and link to data
                     management routine
000094  47 F0 7 03E  B  CL.4  Don't need next instruction
000098  CL.2  EQU  *  Label branched to, if no data
                     management call
000098  BE 17 8 04D  STCM 1,7,77(8)  Save record address
00009C  CL.4  EQU  *  
00009C  D2 4F D 088 1 000  MVC  CARD(80),0(1)  Move record into record
                     variable
0000A2  CL.3  EQU  *  
0000A2  91 C0 2 02C  TM 44(2),X'C0'  Test for errors
0000A6  47 80 7 052  BZ  CL.5  Branch if no errors
0000AA  58 F0 3 01C  L 15,A..IBMRIOD  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 54. In-Line I/O Transmission Statement
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.
### Statement for Record Types F and FB

<table>
<thead>
<tr>
<th>Statement</th>
<th>Record Variable Requirements</th>
<th>ENVIRONMENT Option Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ SET</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>READ INTO</td>
<td>Length known at compile time (max. length if a varying string or area¹)</td>
<td>RECSIZE known at compile time or SCALARVARYING option if varying string</td>
</tr>
<tr>
<td>WRITE FROM (fixed string)</td>
<td>Length known at compile time</td>
<td>RECSIZE known at compile time</td>
</tr>
<tr>
<td>WRITE FROM (varying string)</td>
<td>RECSIZE known at compile time</td>
<td>SCALARVARYING option used</td>
</tr>
<tr>
<td>WRITE FROM Area¹</td>
<td>RECSIZE known at compile time</td>
<td>SCALARVARYING if varying string</td>
</tr>
<tr>
<td>LOCATE A</td>
<td>Length known at compile time (max. length if varying string or area¹)</td>
<td>RECSIZE known at compile time</td>
</tr>
</tbody>
</table>

Figure 56. Conditions Under Which I/O Statements Are Handled In-Line for Record Types F and FB

¹ Including structures whose last element is an unsubscripted area.

**Notes to Figure 56:**
- File type is consecutive buffered with the TOTAL option used.
- 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).
<table>
<thead>
<tr>
<th>Statement for Record Types U, V and VB</th>
<th>Record Variable Requirements</th>
<th>ENVIRONMENT Option Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ SET</td>
<td>None</td>
<td>Not BACKWARDS</td>
</tr>
<tr>
<td>READ INTO</td>
<td>Length known at compile time (max. length if a varying string or area¹)</td>
<td>RECSIZE known at compile time SCALARVARYING option if varying string</td>
</tr>
<tr>
<td>WRITE FROM (fixed string)</td>
<td>Length known at compile time</td>
<td>RECSIZE known at compile time</td>
</tr>
<tr>
<td>WRITE FROM (varying string)</td>
<td>Length known at compile time</td>
<td>RECSIZE known at compile time SCALARVARYING option used</td>
</tr>
<tr>
<td>WRITE FROM Area¹</td>
<td>Length known at compile time (max. length if varying string or area¹)</td>
<td>RECSIZE known at compile time SCALARVARYING if varying string</td>
</tr>
<tr>
<td>LOCATE</td>
<td>Length known at compile time</td>
<td>RECSIZE known at compile time</td>
</tr>
</tbody>
</table>

Figure 57. Conditions Under Which I/O Statements Are Handled In-Line for Record Types U, V, and VB

¹ Including structures whose last element is an unsubscripted area.

Notes to Figure 57:

- File type is consecutive buffered with the TOTAL option used.
- 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).
Appendix C. Stream-Oriented Input/Output

Note on Terminology

In this appendix, 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 (with the exception of GRAPHIC files). Both conversion and record boundary problems are handled automatically.

Note: For stream file use in graphic files, see DBCS continuation rules.

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 58 on page 242. 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-called 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 59 on page 243). 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.
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 58. The Principles Used in Stream I/O
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.

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 60 on page 245. 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.
Figure 60. Simplified Flow Diagram of a Stream I/O Statement
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 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 contents of the SIOCB is as follows:

- Address of source or source locator
- Address of source DED
- Address of target or target locator
- Address of target DED
- Flag bytes
- Address of FCB for file
- Abnormal return address (next statement)
- Save word used by compiler
- Count of items transmitted (Halfword)
- Address of ONCA
- Area present only for GET or PUT STRING, to hold a dummy file control block (27 fullwords).

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 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 “Transmitter Modules” on page 274.

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.

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:

**FCBA** Points at the position reached in current record.

**FREM** Number of unused bytes remaining in the record.

FCBA points to the position reached in the record and enables the director routines to identify from where the next input item must be read, 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 61 on page 248 illustrates the use of FCBA and FREM.

Enqueueing and Dequeueing 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 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.
PL/I STATEMENT:

GET FILE(SYSIN) LIST (A, B);

80 Byte record
In data management buffer

Figure 61. The Use of FREM and FCBA in Recording Buffer Position
Handling the Conversions

Conversions in stream I/O are normally handled by the library conversion package. The conversion package 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 "External Conversation Director Modules" on page 275. 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 60 on page 245 and summarized above.

This appendix 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 62 on page 251. 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,
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. 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 IBMBSTAA. For non-print
files, each item is followed by a single blank. PLITABS is 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 con­
verted 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 trans­
mitter 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. 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 pro­
cessed.

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 trans­
mitter to transmit any information that is still held in the buffer.
The object code produced for a PUT LIST statement is shown in Figure 63 on page 253.

**PUT LIST (A)**

**FLOW DIAGRAM**

1. **Step 1** Compiled code
   - Load address SIOCB
   - Place in parameter list
   - Mark end of parameter list
   - Saves SIOCB in a temporary pointer
   - Set LIST OUTPUT flag
   - Point R1 at parameter list
   - Call stream output initializer

2. **Step 2** Initializing routine IBMBSIO
   - The initialization routine is passed the address of the FCB and the address of the SIOCB.
   - It opens the file if necessary and acquires the first record for print files. If the statement is invalid it calls the error handler. If the statement is valid it places the addresses of the ONCA and the FCB in the SIOCB and returns to compiled code.

**COMPILED CODE & NOTES**

- **LA** 7,200(0,13)
- **ST** 7,768(0,3)
- **OI** 7,6(3),X'80'
- **LA** 1,200(0,13)
- **ST** 1,192(0,13)
- **MVI** 217(13),X'40`
- **LA** 1,72(0,3)
- **L** 15,A,IBMBS1OA
- **BALR** 14,15

---

Figure 62 (Part 1 of 2). Flow of Control through a PUT LIST Statement
Figure 62 (Part 2 of 2). Flow of Control through a PUT LIST Statement
PL/I Source Statements:
DCL A,B STATIC;
PUT LIST (A,B);

* STATEMENT NUMBER 3
00005E 41 70 D 0C8 LA 7,200(0,13) Pick up address of SIOCB
000062 50 70 3 04C ST 7,76(0,3) Store in parameter list
000066 96 80 3 04C OI 76(3),X'88' Mark end of parameter list
00006A 41 10 D 0C8 LA 1,200(0,13) SIOCB pointer
00006E 50 10 D 0C8 ST 1,192(0,13) to temporary pointer
000072 92 40 D 0D9 MVI 217(13),X'40' Set LIST OUTPUT flag in SIOCB
000076 41 10 3 048 LA 1,72(0,3) Point R1 at SIOCB
00007A 58 F0 3 02C L 15,A..IBMBSLOA Branch to initializing module
00007E 05 EF BALR 14,15
000080 41 E0 D 0B8 LA 14,A Pick up address of A
000084 41 F0 3 038 LA 15,DED..A Pick up address of DED..A
000088 58 10 D 0C0 L 1,192(0,13) Restore SIOCB address
00008C 90 EF 1 000 SIM 14,15,0(1) Store addresses in SIOCB
000090 58 F0 3 034 L 15,A..IBMBSLOA Call list-directed director
000094 05 EF BALR 14,15 routine
000096 41 E0 3 050 LA 14,B Pick up address of B
00009A 58 10 D 0C0 L 1,192(0,13) Point R1 at SIOCB
00009E 50 E0 1 000 ST 14,0(0,1) Place address B in SIOCB
0000A2 58 F0 3 034 L 15,A..IBMBSLOA Call list-directed director
0000A6 05 EF BALR 14,15 routine
0000A8 58 10 D 0C0 L 1,192(0,13) Point R1 at SIOCB
0000AC 58 F0 3 030 L 15,A..IBMBSIOT Make terminating call to
0000B0 05 EF BALR 14,15 dequeue on SYSPRINT

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 63. Code Generated for Typical List-Directed I/O Statement

GET LIST Statement

GET LIST statements follow the same sequence, but the transmission is in the
opposite direction. The main differences are:

- If record spanning is involved, the item is assembled in intermediate work-
  space before it is converted.

- 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.

- 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 64 on page 255 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.
GET DATA (A,B);

FLOW DIAGRAM

Step 1
Compiled code

Set up parameter list, call initializer

Step 2
Input initializing module IBMBSII

Set fields in SIOCB

File open?

NO

Call IBMBOCL to open file

YES

Return to compiled code

Step 3
Compiled code

Set up p. list for data director consisting of A(SIOCB) A(SYMTAB,I) A(SYMTAB,J)

COMPILED CODE & NOTES

LA 7,216(0,13) Pick up address of SIOCB
ST 7,84(0,3) Place in parameter list
OI 84(3),X'80' Flag last argument in parameter list
LR 1,7 Get address of parameter list
MVI 234(13),X'84' Set flag DATA INPUT in SIOCB
MVI 234(13),X'11' Reset flag value
LA 14,104(0,3) Set abnormal return
ST 14,240(0,13) Store address in SIOCB
LA 1,80(0,3) Point R1 at parameter list
ST 15,A, .IBMBSIIA Branch to stream
BALR 14,15 initializing module

The input initializing module is passed the address of the SIOCB and the FCB for the file.

It checks the validity of the statement, opens the file and places the address of the FCB in the SIOCB and returns to compiled code

LA 15,216(0,13) Pick up address of SIOCB
ST 15,88(0,3) Place address in parameter list

The parameter list contains addresses of symbol tables and variables already set up in static.

LA 1,88(0,3) Point R1 at parameter list
L 15,A, .IBMBSIIA Call data-directed director module
BALR 14,15 Abnormal locate return address

CL2 EQU *

Figure 64 (Part 1 of 2). Handling a GET DATA Statement
The data directed director module is passed the address of the SIOCB and either a list of symbol table addresses or an address in the symbol table vector.

The module reads in the name, checks that the name read is in the symbol tables passed and if not raises the NAME condition.

When the variable is identified the module places the address of the target and its DED in the SIOCB and calls the list-directed director module passing it the SIOCB.

On return to the data directed module a search is made for the next name and the action continued as from step 4 until a semicolon is reached in the input stream.

Figure 64 (Part 2 of 2). Handling a GET DATA Statement
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 table format is shown in Appendix A, "Control Blocks" on page 119.

The object code produced for a PUT DATA statement is shown in Figure 65 on page 258.

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 (for example, F(3), A(25), etc.), known as data format information.
2. Spacing information (for example, 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.
**PL/I source statements:**
DCL A,B,C;
PUT DATA (A,B,C);

---

**RELEVANT SECTION OF THE STATIC INTERNAL STORAGE MAP**

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000048 00000000</td>
<td>A..DCLCB</td>
<td>Parameter list</td>
</tr>
<tr>
<td>00004C 00000000</td>
<td>A..TEMP</td>
<td>for IBMBSIOA</td>
</tr>
<tr>
<td>000050 00000000</td>
<td>A..TEMP</td>
<td></td>
</tr>
<tr>
<td>000054 00000060</td>
<td>A..SYMTAB</td>
<td>Parameter list</td>
</tr>
<tr>
<td>000058 00000074</td>
<td>A..SYMTAB</td>
<td>for IBMBSDOA</td>
</tr>
<tr>
<td>00005C 00000088</td>
<td>A..SYMTAB</td>
<td></td>
</tr>
<tr>
<td>000060 8500000100000038</td>
<td>SYMBOL TABLE..A</td>
<td></td>
</tr>
<tr>
<td>000088 8500000100000038</td>
<td>SYMBOL TABLE..C</td>
<td></td>
</tr>
<tr>
<td>000074 8000000000000000</td>
<td>SYMBOL TABLE..B</td>
<td></td>
</tr>
<tr>
<td>000088 8500000100000038</td>
<td>SYMBOL TABLE..C</td>
<td></td>
</tr>
<tr>
<td>0000C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**RELEVANT SECTION OF THE OBJECT PROGRAM LISTING**

* STATEMENT NUMBER 3

<table>
<thead>
<tr>
<th>Statement</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00005E</td>
<td>41 70 D 0E8</td>
<td>LA 7,232(0,13)</td>
</tr>
<tr>
<td>000062</td>
<td>50 70 3 04C</td>
<td>ST 7,76(0,3)</td>
</tr>
<tr>
<td>000066</td>
<td>96 80 3 04C</td>
<td>DI 76(3),X'80'</td>
</tr>
<tr>
<td>00006A</td>
<td>18 17</td>
<td>LR 1,7</td>
</tr>
<tr>
<td>00006C</td>
<td>50 10 D 0E0</td>
<td>ST 1,224(0,13)</td>
</tr>
<tr>
<td>000070</td>
<td>92 80 D 0F9</td>
<td>HVI 249(13),X'80'</td>
</tr>
<tr>
<td>000074</td>
<td>92 01 D 0FA</td>
<td>HVI 250(13),X'01'</td>
</tr>
<tr>
<td>000078</td>
<td>41 10 3 048</td>
<td>LA 1,72(0,3)</td>
</tr>
<tr>
<td>00007C</td>
<td>58 F0 3 02C</td>
<td>L 15,A..IBMBSIOA</td>
</tr>
<tr>
<td>000080</td>
<td>05 EF</td>
<td>BALR 14,15</td>
</tr>
<tr>
<td>000082</td>
<td>41 F0 3 0E8</td>
<td>LA 15,232(0,13)</td>
</tr>
<tr>
<td>000086</td>
<td>50 F0 3 050</td>
<td>ST 15,80(0,3)</td>
</tr>
<tr>
<td>00008A</td>
<td>96 80 D 0FB</td>
<td>DI 251(13),X'80'</td>
</tr>
<tr>
<td>00008E</td>
<td>41 10 3 050</td>
<td>LA 1,80(0,3)</td>
</tr>
<tr>
<td>000092</td>
<td>58 F8 3 02B</td>
<td>L 15,A..IBMBSDOA</td>
</tr>
<tr>
<td>000096</td>
<td>05 EF</td>
<td>BALR 14,15</td>
</tr>
<tr>
<td>000098</td>
<td>58 10 D 0E0</td>
<td>L 1,224(0,13)</td>
</tr>
<tr>
<td>00009C</td>
<td>58 F0 3 030</td>
<td>L 15,A..IBMBSIOT</td>
</tr>
<tr>
<td>0000A0</td>
<td>05 EF</td>
<td>BALR 14,15</td>
</tr>
</tbody>
</table>

---

*Figure 65. Typical Data-Directed Code*
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:

- Keeping track of the buffer position, freeing and acquiring intermediate workspace where necessary, and calling the library when a new record is required.
- 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:

- Two short modules, IBMBSE0 and IBMBSEI, that interface with the transmitter and are called by the compiler-generated subroutines when a new record is required.
- A routine, IBMBSEDA, 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 66 on page 260 shows the conditions under which each method is used.
Figure 66. The Use of the Library in Edit-Directed I/O

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 67 on page 261. Figure 68 on page 263 shows the code generated for a GET EDIT statement.
**PUT EDIT (B)(A):**

**FLOW DIAGRAM**

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Compiled code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set up part of SIOCB.</td>
<td>Call initialization routine IBMBSIO</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 2</th>
<th>Initialization routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>File closed</td>
<td>YES</td>
</tr>
<tr>
<td>NO</td>
<td>Call IBMBOCL to open file &amp; call transmitter to get 1st record</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 3</th>
<th>Compiled code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place address of variable, its DED, &amp; DED generated from format item, in SIOCB</td>
<td>Call IELCGOA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 4</th>
<th>IELCGOA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Will item span or require new record?</td>
<td>YES</td>
</tr>
<tr>
<td>NO</td>
<td>Set 'VDA' flag in SIOCB. Get VDA and set as address of target.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NOTES</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LA 7,216(0,13)</td>
<td>Pick up address of SIOCB</td>
</tr>
<tr>
<td>ST 7,840(0,3)</td>
<td>Place in parameter list</td>
</tr>
<tr>
<td>GI 84(3), 'X'80'</td>
<td>Mark end of parameter list</td>
</tr>
<tr>
<td>LR 1,7</td>
<td>Point R1 at SIOCB</td>
</tr>
<tr>
<td>ST 1,208(0,13)</td>
<td>Save in temp</td>
</tr>
<tr>
<td>MVI 233(13), 'X'20'</td>
<td>Set EDIT OUTPUT flag</td>
</tr>
<tr>
<td>LA 1,800(0,3)</td>
<td>Point R1 at parameter list</td>
</tr>
<tr>
<td>L 1,5,A, IBMBSIOA</td>
<td>Branch to initialization routine</td>
</tr>
<tr>
<td>BALR 14,15</td>
<td></td>
</tr>
</tbody>
</table>

Test if file is open, and open if necessary, calling transmitter to locate record. Place address of ONCA and FCB in the SIOCB. Check statement validity.

<table>
<thead>
<tr>
<th>Step 3</th>
<th>Compiled code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA 14,B</td>
<td>Get address of data</td>
</tr>
<tr>
<td>LA 15,0,ED. B</td>
<td>Get address of DED. B</td>
</tr>
<tr>
<td>L 1,208(0,13)</td>
<td>Get SIOCD address</td>
</tr>
<tr>
<td>STM 14,15,0(1)</td>
<td>Place addresses of B and DED. B in SIOCB</td>
</tr>
<tr>
<td>LA 14,15,0,1(3)</td>
<td>Get address of FED</td>
</tr>
<tr>
<td>L 7,A, IELCGOA</td>
<td>Branch to compiler-generated subroutine</td>
</tr>
<tr>
<td>BALR 5,7</td>
<td></td>
</tr>
</tbody>
</table>

Acquire VDA for item if necessary. Either if there is no room in current record, or, if the converted item will span the record boundary.

Figure 67 (Part 1 of 2). Edit-Directed Statement with Matching Data and Format Lists
Carry out conversion either in-line or by calling library module.

Call IELCGOB

Step 5
Compiled code or conversion routine

Call IELCGOB

Step 6
IELCGOB YES

Update buffer control fields
Handle housekeeping

Was a VDA used?

NO

Clear 'VDA' flag and IBMBSEDB flag

Yes

Call IBMBSEGDA Call transmitter and free VDA

Update FREM, FGBA, and FCNT

Return to compiled code

Step 7
Compiled code

Continue from STEP 3 with next item, if any
When complete make terminating call to IBMBSIOT

Continue as necessary
When complete call termination routine to dequeue on SYSPRINT

L 15, A. IBMBSAOA
BALR 14, 15

Call output conversion director (A-format)

L 7, A. IELCGOH
BALR 6, 7

Call compiler-generated director routine

L 1, 208 (0, 13)
Get SIOCB address

Figure 67 (Part 2 of 2). Edit-Directed Statement with Matching Data and Format Lists
**PL/I source statements:**

```plaintext
DCL A,B;
GET EDIT (A,B) (F(3), X(8));
```

---

**Figure 68. Code Generated for an Edit-Directed Statement with Matching Data and Format Lists**
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:

IBMBSPL  Library routine for SKIP, PAGE, and LINE formats and options.
IBMBSXC  Library routine for X and COLUMN formats.
IELCGOC  Compiler-generated subroutine for X output items that do not span a record boundary.
IELCGIX  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 appendix.)

Matching and Nonmatching 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 not possible 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 sequence used for matching and non-matching data and format lists are shown in Figure 69 on page 265.
MATCHING LISTS
PUT EDIT (I, NAME, ACT. NO)
(F (3), X (3), A (15), X (3), 'PZZZ9');

UNMATCHING LISTS
PUT EDIT (AB, C, D) ((N) F (3), SKIP, A (4));

Figure 69. Code Sequences Used for Matching and Nonmatching Data and Format Lists
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 IBMSTAB 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 IBMSTAB, which holds the default tab setting. When the routine is loaded, the address of entry point IBMSTAB 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.

Input and Output of Complete Arrays

When transmitting complete arrays, it is not economical 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 **transferred 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 as soon as 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, IBMBSEIT 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, IBMBSCVA. 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 cannot 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.

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 the 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. As shown in Figure 70 on page 269, 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 difference in housekeeping between string and file operations is 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.

Figure 70. The Current Buffer Pointer FCBA and FCPM, the Copy Pointer, Keep Track of the Data to be Copied
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 '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 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 IBMBSISA is called to initialize a statement. The library director routines and the compiler-generated subroutines test this flag, and call IBMBSIS if necessary.

The Conversational System and Conversation Files
When using a conversational system, the PL/I programmer can attach his 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 the conversational system using the appropriate macro instructions, or simulations of them for CMS, to effect the input and output. They also poll for attention interrupts. 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 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, IBMSPC. 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. For CMS it uses a simulation of TPUT. 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. For CMS it uses a TGET simulation.

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 the 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)

The types of modules are dealt with below.

Initializing Modules

Initializing modules initialize the stream I/O statement. There are two of these modules:

IBMBSSII Input initializer
IBMBSSIO Output initializer

A further module is used for string handling, which is listed under “Miscellaneous Modules” on page 275.
IBMBSII is discussed in “The COPY Option” on page 268, while IBMBSIO is described under “PUT LIST Statement” on page 249.

**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:

IELCGIX 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 the VDA if one was used, updating the COUNT built-in function value, and calling IBMBSEIT if TRANSMIT has been raised.
For output:

IELCGOG  Provides the address of the target of an edit-directed data item.

IELCGOH  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

IBMSBPL  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

IBMSBXC  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
IELCGOCA X items, in edit-directed output, that do not span a record boundary.

External Conversation 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
- IBMBSF1 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
Index

A
ABEND dump 90
debbuging with 91
EPIE 98
extended program interrupt element 98
interrupt 90
interrupt caused 75
interrupts 98
NOSPIE used 75
NOSTAE used 75
PIE 98
program check exit 90
program interrupt element 98
User Exit 75
abnormal GOTO statement
changing CHECK enablement during 42
from an event I/O ON-unit 42
library subroutine IBMBPGO 39
out of SORT exit routine 42
abnormal locate return block 220
access method
record I/O 209
stream I/O 241
activating blocks 33
address constants
O-type 29
addressing
automatic variables 27
allocated storage in DSA 27
allocated storage in VDA 27
beyond the 4K limit 28
based variables 28
beyond the 4K limit 28
controlled variables
pseudo-register vector 27
files via DCLCB and PRV 218
static variables 28
beyond the 4K limit 28
temporary variables 27
allocated storage in DSA 27
allocated storage in VDA 27
beyond the 4K limit 28
addressing beyond 4K limit 28
aggregates (see also arrays and structures)
arrays of structures 32
descriptor descriptor 122
locator 124
aggregates (see also arrays
structures) 31
ALLOCATE command
using for dumps 75
APAR (Authorized Program Analysis Report) (continued)
CMS terminal session log 117
compiler listing 117
JCL listing 117
linkage editor 118
machine-readable information to submit 116
materials accompanying 115
run-time dump 118
source information 116
area
control block 121
descriptor 120
locator/descriptor 120
argument lists 43
DO-LOOPS, use of 45
in static storage 43
passed by calling routine 43
setting up 44
arrays
assignments 32
boundaries 31
descriptor descriptor 31
descriptor 125
implementation of 31
interleaved 32
multipliers 31
of structures 32
program control data 31
virtual origin 31
automatic variables
addressing beyond the 4K limit 28
definition 27
initialization of 33
storage in DSA 27
storage in VDA 27
B
backchains
dynamic 36
static 36
base registers
DSA pointer 25
program base 25
static base 25
TCA pointer 25
based variables 28
blocks
activating 33
terminating 33
books, OS PL/I Version 2 iv
branches, rationalization of 54
buffer control fields (stream I/O) 247
buffer pointers (stream I/O) 247

C
C format item DED 139
CALL statements 38
CICS
  ABEND 111
  appendage 128
  modules 62
  run-time environment 111
  user exit 111
CLOSE statement
  compiler output 227
  general 205
closing files
  explicit closing 227
  implicit closing 206, 227
  library subroutines 227
COLUMN format item 274
common expression, elimination of 47
commoning
  for optimization 54
compile-time options
  AGGREGATE 14
  documentation provided by 3
  ESD 14
  LIST 14
  MAP 14
  OFFSET 14
  problem determination, helpful in 3
  SOURCE 14
  STORAGE 14
  SYSTEM 112
  TEST 3
compile-time problem determination 5
  chart 5
  chart index 5
compiler function 3
compiler output 11–55
  control sections 11
  dummy sections 12
    pseudo-register vector 12
  ESD records 11
  relocatable object module 11
  RLD records 11
  TXT records 11
  constants 11
    machine instructions 11
compiler-generated subroutines 259
  IELCGIX 264
  IELCGOC 264
  purpose of 46
conditions
  name abbreviations in a dump
  error code field table 86
conditions (continued)
  name abbreviations in dump 84
  consecutive buffered files 208
  constants 22
  constants pool 22
  contents of listing information 14
control block
  locating in dump
  quick guide 108
control blocks
  array descriptor 31
  for optimization
    commoning 54
  formats 119
  in a PL/I environment 4
  non-VSAM section 166
  structure descriptor 31
  VSAM section 167
control format information 257
  DED 139
control sections 11–55
  PLICOUNT 12
  PLIFLOW 11
    IBMELFL, trace module 12
  PLIMAIN 11
  PLISTART 11
  program 11, 23
  static external 12
    static storage map 15
  static internal 11, 22
    static storage map 15
controlled variable block 130
controlled variables
  control block 130
  header information 28
  pseudo-register vector 27, 29
controlling the flow of execution
  non-consecutive 32
    epilog code 38
    prolog code 33
conversational files 270
conversational transmitter modules
  IBMBASIC 271
  IBMBSOC 271
  IBMBSPC 271
conversion
  stream I/O 249
CONVERSION condition
  in stream I/O 267
COPY option
  in stream I/O 268
COUNT function 268
CSSF (Customer Software Support Facility) 113
Customer Software Support Facility
  See CSSF (Customer Software Support Facility)
data element descriptors (DEDs) 132
data format item 257
data list matching 260
data management event control block
   non-VSAM 156
   VSAM 167
data-directed I/O 254
DATAFIELD built-in function 267
DCLCB (declare control block)
   format 140
   general 210
DD statement
   using for dumps 75
debugging a dump
   suggested code 80
declare control block (DCLCB)
   format 140
   general 210
dedicated registers 25
   DSA pointer 25
   program base 25
   static base 25
   TCA pointer 25
DELETE statement 203
director routines
   definition 241
   in stream I/O 249
DO-loops
   accessing array elements
      example of 53
   loop control variables 52
   example of 45
   nested 45
documentation, OS PL/I Version 2  iv
dummy FCB 30
dummy ONCA 172
dummy sections
   constants 11
   machine instructions 11
   pseudo-register vector 12
dump
   ABEND dump 98
   avoiding recompilation 81
   calling trace 101
   chains 105
   tasking appendage, in 105
debugging
   where to start 87
debugging procedures 87–91
debugging with 75–111
   ABEND dump 91
   ON-unit call procedure 89
   storage overlay 88
   system ABEND dump 90
dump (continued)
   DSA
      associating with block 101
   DSA addresses 96
   DSA Chain 98
   dynamic storage area (DSA)
      finding 104
      following back-chain in dump 99
      register save area in 103
   finding variables in
      area variables 108
      automatic variables 107
      based variables 108
      controlled variables 107
      static variables 107
   forward chain in DSAs 104
   GOSTMT option 99
   heap
      free-area chain 107
      storage chain 107
   hexadecimal 86
   housekeeping information 99
   IBMBER
      DSA contents after interrupt 95
   initial storage area (ISA)
      free-area chain 102
   interrupts
      register contents at point 98
   library module error in 104
   locating information in 92
   control blocks and fields 108
   list of contents 92
   PL/I dump 93
   multitasking in 111
   OFFSET option 101
   ON-unit
      statement number and address 93
      type of error 93
   parameter list
      contents 104
   register contents 94
   interrupt 94
   interrupt in library routine 94
   program check interrupt 94
   register save areas
      finding and interpreting 102
   register usage 102
   registers
      point of interrupt, at 98
   SNAP dump
      PSW 88
   stand-alone 75
      locating information in 99
   statement number 99
   offset table, in 101
   static back chain in 104
   system ABEND 90
   task appendage 107
   TCA in 107
dump (continued)
  task relationships 105
    parent 105
    sibling 105
    subtasks 105
  task variable in 103
  TCA 98
  TCA addresses 97
  timestamp (TSTAMP) 97
  trace information
    not generated 96
    use of block option 87
  dynamic backchain 36
  dynamic ONCB 175
  dynamic storage area (DSA) 57
    addresses in dump 96
    allocation 57
    associating DSA with block in dump 101
    chaining of 97
    chaining when multitasking 111
    contents for compiled code DSA 37
    current 25
    finding main procedure DSA in dump 104
    following back-chain in dump 99
    format and function 142
    forward chain in dump 104
    IBMBERR's DSA in dump 102
    register save area in 103

E
  E format item DED 138
  edit-directed I/O 257–265
    code generated 263
    compiler-generated subroutines 259
    data format items 257
    FED 257
    format DED 257
    format list 264
    format option handling 264
    GET statement 257
    handling control format items 264
    library director modules 273
    matching data and format lists 261, 264
    nonmatching data and format lists 264
    PUT statement 257
    typical statement 260
    use of library in 260
    elimination of unreachable statements 50
    enablement status 40
    end of file 233
    END statement 32
    ENDFILE condition
      in record I/O 233
      in stream I/O 267
    entry data control block 146
    entry points
      load module 11

entry points (continued)
  main procedure 11
  PLISTART 11
    IBMBPIR, address of 12
  entry variables
    entry data control block 146
  ENVB (see environment control block)
  environment
    definition 4
    environment control block (ENVB) 212
      format 147
      record I/O 210
      stream I/O 246
  epilog code
    example of 38
    FINISH condition 38
  error code
    field lookup table 86
    list of 86
  error handling
    during execution 86
  error identification
    using the dump 75–111
  ERROR ON-unit and dumps 79
  error-handling 90
  ESD records
    external addresses 11
  EV (event variable) 220
  event I/O 224
  EVENT option 224
  event variable (EV) 220
  event variables
    control block format and function 151
    exclusive I/O 226
    external conversion director modules 275

F
  F format item DED 138
  FCB (see file control block)
    FCBA field in FCB 247
    FCPM field in FCB 268
    FECB (fetch control block) 163
    FEDs (format DEDs) 257
    FFFT field in FCB 231
    FERM field in FCB 233
    FERM field in FCB 231
    fetch control block (FECB) 163
    FETCH statements
      control block
        FECB 163
    fields
      locating in dump 108
      file control block (FCB) 205
      FCBA field 247
      FCPM field 268
      FFFT field 231
      FERM field 233
file control block (FCB) (continued)
FERM field 231
fields for buffer operation 247
format and function 152
FREM field 247
general description 211
record I/O 160
stream I/O 161
file declaration statement 212
file information in a dump
block option 87
buffer contents 85
hexadecimal dump
in multitasking 86
file organization
stream I/O 246
FILEDEF command
using for dumps 75
files
addressing 29
closing 227
conversational 270
declaration 203, 210
explicit opening 213
implicit opening 229
opening 213, 246
record variable 219
fixed-point data
DED 134
floating-point data
DED 134
floating-point register
usage 26
flow of control 32–42
format DEDs (FEDs) 257
format element descriptor (FED)
format and function 138
format items 264
format list matching 264
formatting modules 264
FREM field in FCB 247
function references
example of 39

G
G format item DED 138
GET DATA statement
handling 256
GET LIST statement 253
GOTO statement
-only ON-units 42
abnormal 42
interpretive code for 39
interpretive routines 42
label variable 41
out of block
example of 41

H
heap
free-area chain 107
storage chain 107
hexadecimal dump
in multitasking 86

I
I/O
conditions for handling statements in-line 239
event 224
exclusive 226
in-line
control blocks 234
error conditions 234
executable instructions 234
implicit open 234
transmission statement 236
library-call 218, 229
record 203–240
stream 241–275
IBM programming support 115
IBMIEER
DSA contents after interrupt 95
IBMBERR
DSA in dump 95
dump address 83
IBMBOCA 207
IBMBOCL 205, 207, 213
IBMOPA 207
IBMOPB 207
IBMOPC 207
IBMOPD 207
IBMOPF 207
IBMOPZ 207
IBMBPOG (interpretive GOTO) 39
IBMBPAA (regional seq output trans) 207
IBMBRAAB (regional seq output trans) 207
IBMBRBC (regional seq output trans) 207
IBMBRBD (regional seq output trans) 207
IBMBRBE (regional seq output trans) 207
IBMBRBF (regional seq output trans) 207
IBMBRAG (regional seq output trans) 207
IBMBRAH (regional seq output trans) 207
IBMBRAI (regional seq output trans) 207
IBMBRRA (regional seq in/upd trans) 207
IBMBRBB (regional seq in/upd trans) 207
IBMBRBC (regional seq in/upd trans) 207
IBMBRBD (regional seq in/upd trans) 207
IBMBRBE (regional seq in/upd trans) 207
IBMBRBF (regional seq in/upd trans) 207
IBMBRBB (regional seq in/upd trans) 207
IBMBRCA (unbuffered consec trans) 207
IBMBRCB (unbuffered consec trans) 207
IBMBRCC (unbuffered consec trans) 207
IBM RCD (unbuffered consec OMR) 207
IBM RCE (unbuffered consec associated file) 207
IBM RDA (regional direct non-exclusive trans) 207
IBM RDB (regional direct non-exclusive trans) 207
IBM RDC (regional direct non-exclusive trans) 207
IBM RDD (regional direct non-exclusive trans) 207
IBM PEA (record I/O error module) 208
IBM REB (record I/O error module) 208
IBM RCE (record I/O error module) 208
IBM REF (record endfile module) 208
IBM RO (record I/O interface) 205, 219
IBM RJA (indexed seq in/upd trans) 207
IBM RJB (indexed seq in/upd trans) 207
IBM RK (indexed direct non-exclusive trans) 207
IBM RKB (indexed direct non-exclusive trans) 207
IBM RKC (indexed direct non-exclusive trans) 207
IBM RLA (indexed sequential output) 208
IBM RLB (indexed sequential output) 208
IBM ROA (buffered non-spanned trans) 208
IBM ROB (buffered non-spanned trans) 208
IBM ROC (buffered non-spanned trans) 208
IBM ROD (buffered non-spanned trans) 208
IBM ROE (buffered input spanned trans) 208
IBM ROF (buffered output spanned trans) 208
IBM ROG (buffered update spanned trans) 208
IBM ROH (buffered consecutive OMR) 208
IBM ROI (buffered associated file) 208
IBM RTP (teleprocessing input trans) 208
IBM RVA (VSAM ESDS transmitter) 208
IBM RVG (VSAM KSDS sequential output) 208
IBM RVRI (RRDS) 208
IBM RVVM (VSAM KSDS other operations) 208
IBM RXA (exclusive regional direct upd/input trans) 208
IBM RXB (exclusive regional direct upd/input trans) 208
IBM RXC (exclusive regional direct upd/input trans) 208
IBM RXD (exclusive regional direct upd/input trans) 208
IBM RYA (exclusive indexed direct upd/input trans) 208
IBM RYB (exclusive indexed direct upd/input trans) 208
IBM RYC (exclusive indexed direct upd/input trans) 208
IBM RYD (exclusive indexed direct upd/input trans) 208
IBM SAI 275
IBM SAO 275
IBM SCI 275
IBM SCO 275
IBM SCP (copy module) 272, 275
IBM SCV 275
IBM SDI 273
IBM SDO 273
IBM SED 273
IBM SEI 273
IBM SEO 273
IBM SFI 275
IBM SFO 275
IBM SCI 271, 275
IBM SII 272
IBM SIO 272
IBM SIS 269, 275
IBM SII 273
IBM SLO 273
IBM SOC 271, 275
IBM SOF (stream output file trans) 208, 274
IBM SOU (stream output file trans) 208, 274
IBM SOV (stream output file trans) 208, 274
IBM SPC 271, 275
IBM SPI 275
IBM SPL (formatting module) 264
IBM SPO 275
IBM STF (stream output print file trans) 208, 274
IBM STI (stream input file trans) 208, 274
IBM STU (stream output print file trans) 208, 274
IBM STV (stream output print file trans) 208, 274
IBM XCF 264, 274
IBM CTI (stream input file) 208
IBM CSTP (stream output file) 208
IBM SPL 274
IELCGB (test for '0' bits) 47
IELCGBO (test for '1' bits) 47
IELCGB (compare long bit) 46
IELCGCL (compare long) 46
IELCGIA (stream I/O input) 46
IELCGIB (stream I/O input) 46, 273
IELCGIX (stream I/O input) 264, 273
IELCGMV (move long) 46
IELCGOC (stream I/O output) 46, 264
IELCGOG (stream I/O output) 46, 274
IELCGOH (stream I/O output) 46, 274
IELCGON (dynamic ONCB chaining) 47
IELCGRV (revert VAD chaining) 47
implicit close in record I/O 208
implicit open
record I/O 205
stream I/O 247
in-line calls
implicit open 234
in-line I/O 205
initial storage area (ISA)
following free-area chain in dump 102
initialization
of the PRV 30
stream I/O subroutines 272
input/output control block (IOCB) 164
interpretive GOTO routine IBMBPGO 39
### interrupt
- ABEND, causing 90
- error-handling 90
- library routine, in 94
- program check 94

### invariant expressions
- elimination of 49

### IOCB (input/output control block) 164

### J
- job control language (JCL)
  - APAR listing 117

### K
- KD (key descriptor) 219
- key descriptor (KD) 219
- key variable 219

### L
- label data control block 170
- label data format 170
- label variables
  - errors when using 41
  - general description 41
  - in GOTO statements 41
    - with NOOPTIMIZE 41
    - with OPTIMIZE (TIME) 41
- library calls
  - addressing a subroutine 44
  - example of 44
  - general 43
  - level of optimizing used 43
  - naming conventions 43
  - resident library
    - bootstrap routines 43
    - setting up argument lists 44
    - within TCA 45
- library module 104
- library register usage 26
- library routine
  - IBMBSPL 264
  - IBMBSXC 264
- library subroutines
  - in record I/O 206
  - in stream I/O 274
  - naming conventions 62
- library workspace (LWS)
  - format and function 171
- library-call I/O 205, 219
  - implicit open 229
- LINE format option 264
- link editing, definition 4
- linkage editor
  - APAR listing 118

### list-directed I/O 249
### listing conventions 14, 18
### load module
  - contents
    - compiler output 11
    - entry point 11
- LOCATE statement 203, 220
- locators
  - aggregate locator format and function 124
  - area locator/descriptor format and function 120
  - string locator/descriptor format and function 181
- loops
  - rationalization of program branches 54
- LWS (library workspace)
  - format and function 171

### M
- manuals, OS PL/I Version 2 iv
- modification of DO-loop control 51
- module, object 11
- movement of expressions out of loop 49
- multitasking
  - in dumps 111

### N
- NAME condition
  - in stream I/O 267
- naming conventions
  - of library modules 62
- non-VSAM section of control blocks
  - data management event control block 166
- null value 31

### O
- object module 11
- object program listing
  - contents 18
  - DECLARE statements 18
  - entry point 18
  - executable statements 18
  - LIST option 18
- offset table
  - dump, in 101
- offsets
  - null value 31
- ON communications area (ONCA)
  - dummy 172
  - finding relevant ONCA in dump 102
  - following chain of ONCAs in dump 102
  - format and function 172
- ON control block (ONCB)
  - dynamic 175
  - format and function 175
  - static 175
ON-units 42
event I/O 42
GOTO-only 42
open control block (OCB) function 210
OPEN statement
compiler output 213
execution 213
opening files
record I/O 213
stream I/O 246
optimization
branching around redundant expressions 53
compiler approach to 47
compiler options
NOOPTIMIZE 47
OPTIMIZE (TIME) 47
examples of
branching around redundant expressions 53
elimination of common expressions 47
elimination of unreachable statements 50
modification of DO-loop control variables 51
movement of invariant expressions 49
simplification of expressions 51
movement of expressions out of loops 49
using common constants and control blocks 54
commoning, example of 54
options
FLOW 12
LIST 18
MAIN 12
OS PL/I Version 2
publications 4
output from compiler 13

PLDUMP (continued)
how to call 78–79
obtaining 78, 79
tasking option 79
PLIFLOW 13
PLIMAIN
format 177
PLISTART
parameter list 178
PLITABS 266
PLITEST 3
calling PLDUMP 81
pointer variables
misuse of 88
pointers
COPY option 268
how held 27
null value 31
preprocessing
suspected failure in
listing, for APAR 116
preprocessor function 3
print files, formatting for 266
procedure block
epilog code 33
prolog code 33
PROCESS statement
LIST option 18
program base register 25
program check exit 90
program control data
data aggregates 31
arrays 31
structures 31
program control section
general registers 25
dedicated 25
work 25
program listing information 14
program management area
PRV, location of 30
prolog code
acquiring a dynamic storage area (DSA) 33
example of 34
prompting 271
pseudo-register vector (PRV)
addressing controlled variables 29
addressing files 29
file control blocks (FCBs) 29
ALLOCATE statement 29
initialization of 30
location of 30
O-type address constants 29
use of 30
PSP (Preventive Service Planning) 113
PTF (Program Temporary Fix) 113
publications, OS PL/I Version 2 iv
PUT statement 249

R
RCB (request control block) 219
RD (record descriptor) 180, 219
READ statement 203
recompiling
avoiding 81
record descriptor (RD) 180, 219
record I/O
control blocks generated 210
definition 203
table of 232
table used in 232
error handling
fields used in 232
error modules 231
FCB 217
implementation 204
access method 209
CLOSE statement 205
fields for the 210
fields for the fields used for 210
file declaration 203
file input/output 205
implicit close 206
implicit open 205
OPEN statement 205
transmission statement 205
in-line 233
interface routine (IBMIRIO) 219
library subroutines used 207
overview 238
VSAM data sets 214
redundant expressions
branching around 53
each a 54
example of 54
registers 24-26
request control block (RCB) 219
description 211
RETURN statement
chainback to DSA 39
each a 39
example of 39
REWIRE statement 203
RLD records
internal addresses 11
run-time
APAR dump 118
run-time organization
run-time problem determination 63
chart 63
chart index 63
running
definition 4
S
SIOCB (stream I/O control block) 246
SIZE 88
SKIP format option 264
SNAP dump
PSW 88
Software Search Facility
See SSF (Software Support Facility)
squashed mode 272
SSF (Software Support Facility) 113
MVS INFO/ACCESS 113
search arguments
formulating 113
keywords 113
standardized keywords 114
using INFO/ACCESS with 113
using RETAIN with 113
STAE work area
finding 98
stand-alone dump 75
locating information in 99
statement number
in dump 99
of error 99
statement number table
locating in dump 110
static back-chain 104
static base register 25
static control sections
contents 15
static internal control section
addresses 22
entry points 22
external procedures 22
label contents 22
library modules 22
branch tables 22
constants pool 22
contant values 22
data element descriptors (DED) 22
ONCBs 22
symbol table address vector 22
static variables 22
static ONCB 175
static storage map
example of listing 17
static external control sections 15
static internal control sections 15
static variables 28
addressing 28
beyond the 4K limit 28
storage
overall use 61
storage overlay 88
pointer variables, misuse of 88
variable-length files 88

LY27-9528-0 © Copyright IBM Corp. 1985, 1987
stream I/O
  buffer control fields 247
  built-in functions 268
  control block (SIOCB) 246
  conversation files 270
  conversational system 270
    formatting module IBMBSPC 271
    input transmitter IBMBSCIC 271
    output transmitter IBMBSOC 271
COPY option 268
COUNT function 268
DATAFIELD function 268
director routines 241
end of file 267
file handling 246
  conversions 249
    data-directed GET and PUT statements 254
    edit-directed GET and PUT statements 257
    list-directed GET and PUT statements 249
file opening 246
format items 264
format lists 264
formatting for print files 266
handling format options 266
input and output of complete arrays 266
ONCHAR function 268
ONSOURCE function 268
operations 243
PL/I conditions in 266
principles used in 242
simplified flow diagram 245
STRING option 269
summary of subroutines used 272
  conversational modules 275
  director modules 273
  external conversation director modules 275
  formatting modules 274
  initializing modules 272
  transmitter modules 274
STRING option
  completing string-handling operations 270
  housekeeping routine (IBMBSIS) 269
  implementation 269
STRINGRANGE 88
strings
  DED 133
  descriptor 181
  FED 139
  locator/descriptor 181
STRINGSIZE 88
structures
  assignments 32
  descriptor 182
  implementation of 31
  of arrays 32
subpool 98
subroutines
  compiler-generated (continued)
    compiler-generated (continued)
      purpose of 46
SUBSCRIPTRANGE 88
symbol table (SYMTAB) 183
SYMTAB (symbol table) 183
SYSTEM 112
T
  tab table 266
task
    finding the relationship between in a dump 105
task communications area (TCA) 25
      flags 193
      format and function 190
      implementation appendage 196
task variable (TV)
      format and function 202
tasking appendage (TTA) 105
TCA
    addresses in dump 97
    TCA (see task communications area)
temporaries (see temporary variables) 27
temporary variables
    addressing beyond the 4K limit 28
    storage in DSA 27
    storage in VDA 27
terminating blocks 33
TIA (TCA implementation appendage) 196
timestamp (TSTAMP)
    in dump 97
TOTAL option 205
trace
    following calling in dump 101
trace information
    not generated in dump 96
trace information in a dump
      condition names 84
      hexadecimal 83
      IBMBERR 83
      multitasking programs 82
      ON-units 82
trace information in dumps 82
transient open routines 213
transmission statements
  compiler output 219
    definition 203
  ENDFILE routine 233
  error conditions in 229
  execution of 222
  general error routines 233
  in record I/O 205
  TRANSMIT condition 233
  use of EVENT option 224
TRANSMIT condition 233
  in stream I/O 267
transmitter interface module (IBMBRIO) 219
transmitter modules
  record I/O 207
  stream I/O 271
TTA (TCA tasking appendage)
  function and format 200
TV (task variable)
  format and function 202
TXT records 11

U
unexpected end of file
  in stream I/O 267
UNLOCK statement 203
unreachable statements
  elimination of 50
use of storage 81
User Exit
  ABEND dump 75
  Invoking 111
  Name 111

V
variable data area (VDA)
  description 27
variable-length files 88
variables
  area
    in a dump 108
    automatic 27, 107
    in a dump 107
  based 28
    in a dump 108
  controlled 27
    in a dump 107
  label 170
  locating in dump 110
  pointer 28
  static 28, 107
    in a dump 107
  task 103
  temporaries 27
variables, handling and addressing 27
VDA (see variable data area)
VSAM data sets
  opening 214
VSAM section of control blocks
  data management event control block 167

W
WAIT statement
  label variable
    example of 41
    with NOOPTIMIZE 41
    with OPTIMIZE (TIME) 41
  termination of 42
work registers 25
  floating point registers 26
  library registers 26
WRITE statement 203

X
X format item 274
OS PL/I Version 2 Problem Determination

This manual is part of a library that serves as a reference source for system analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: Do not use this form to request IBM publications. If you do, your order will be delayed because publications are not stocked at the address printed on the reverse side. Instead, you should direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

If you have applied any technical newsletters (TN Ls) to this book, please list them here: ___________________________

Chapter/Section ____________________________________________________ Page No. ____________________

Comments:

If you want a reply, please complete the following information.

Name ____________________________________________ Phone No. (____) __________________________

Company ____________________________________________________________________________________

Address ________________________________________________________________________________________

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 or you may mail directly to the address in the Edition Notice on the back of the title page.)
OS PL/I Version 2 Problem Determination

This manual is part of a library that serves as a reference source for system analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author’s department for whatever review and action, if any, are deemed appropriate.

Note: Do not use this form to request IBM publications. If you do, your order will be delayed because publications are not stocked at the address printed on the reverse side. Instead, you should direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

If you have applied any technical newsletters (TNLs) to this book, please list them here: __________________________

Chapter/Section ______________________________________________________________ Page No. __________________________

Comments:

If you want a reply, please complete the following information.

Name ________________________________________ Phone No. (____) ______________________

Company _______________________________________________________________________

Address _______________________________________________________________________

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 or you may mail directly to the address in the Edition Notice on the back of the title page.)
OS PL/I Version 2 Problem Determination

This manual is part of a library that serves as a reference source for system analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author’s department for whatever review and action, if any, are deemed appropriate.

Note: Do not use this form to request IBM publications. If you do, your order will be delayed because publications are not stocked at the address printed on the reverse side. Instead, you should direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

If you have applied any technical newsletters (TNLs) to this book, please list them here: _____________________________

Chapter/Section ___________________________________________ Page No. ___________________________

Comments:

If you want a reply, please complete the following information.

Name ___________________________ Phone No. ( ___ ) _____________

Company ___________________________________________________________

Address ____________________________________________________________

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 or you may mail directly to the address in the Edition Notice on the back of the title page.)