z/OS

MVS Program Management: Advanced Facilities

SA22-7644-14

Note
Before using this information and the product it supports, read the information in “Notices” on page 363.

This edition applies to version 1, release 13, modification 0 of IBM z/OS (product number 5694-A01) and to all subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces SA22-7644-13.

© Copyright IBM Corporation 1991, 2012.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures ix

Tables xi

About this document xv
Required product knowledge xv
Required publications xv
Related publications xvi
Referenced publications xvi
Notational conventions xvi
Where to find more information xviii
Information updates on the web xviii
The z/OS Basic Skills Information Center xviii

How to send your comments to IBM xix
If you have a technical problem. xix

Summary of changes xxi
Changes made in z/OS Version 1 Release 13, as updated September 2012 xxi
Changes made in z/OS Version 1 Release 13 xxi
Changes made in z/OS Version 1 Release 12 xxi
Changes made in z/OS Version 1 Release 11 xxii

Chapter 1. Using the binder application programming interfaces (APIs) . . . 1
Understanding binder programming concepts 1
Symbol names in API calls 5
Program management support of class attributes 5
Organization of data in the program module. 7
Version number in an API call 8
Binder dialogs. 8
Processing intents. 10

Chapter 2. IEWBUFF - Binder API buffers interface assembler macro for generating and mapping data areas 13
Using the IEWBUFF macro 13
FREEBUF: Free buffer storage 14
GETBUF: Get buffer storage 15
INITBUF: Initialize buffer header 16
MAPBUF: Map buffer declaration 16

Chapter 3. IEWBIND - Binder regular API functions 19
Invoking the binder API 19
Setting the invocation environment 19
Loading the binder 19
Invoking the binder using the macros. 20
Invoking the binder without the macros 20
Binder API common return and reason codes 21
Coding the IEWBIND macro 23
Defining varying character strings 23
Defining section names 23
Defining parameter lists. 23
Setting null values. 23
IEWBIND function reference 24

© Copyright IBM Corp. 1991, 2012

ADDA: Add alias 28
ALIGNT: Align text. 30
ALTERW: Alter workmod 32
AUTOC: Perform incremental autocall 36
BINDW: Bind workmod 37
CREATEW: Create workmod 41
DELETEW: Delete workmod 42
DLLR: Rename DLL modules 44
ENDD: End dialog. 46
GETC: Get compile unit list 47
GETD: Get data 50
GETE: Get ESD data 52
GETN: Get names 56
IMPORT: Import a function or external variable 59
INCLUDE: Include module. 61
INSERTS: Insert section 68
LOADW: Load workmod 69
ORDERS: Order sections 72
PUTD: Put data 73
RENAME: Rename symbolic references 78
RESETW: Reset workmod 79
SAVEW: Save workmod 81
SETL: Set library 86
SETO: Set option 88
STARTD: Start dialog 91
STARTS: Start segment 98
Binder API reason codes in numeric sequence 99

Chapter 4. IEWBFDAT - Binder Fast data access API functions 109
Using the fast data access service 109
Environment 110
Parameter descriptions 110
Buffer 111
Class 112
Count 112
Cursor 112
DCBptr 113
DDname 113
DEptr 113
Eptoken 113
Member 114
Mtoken 114
Path 114
Retcode 114
Rsncode 114
Section 114
The Request Code interface 114
Function code usage summary 115
Common Parameters 116
Optional parameters 116
SB - Starting a session with a BLDL identifier 117
SJ - Starting a session with a DD name or path 118
SQ - Starting a session with a CSVQUERY token 118
SS - Starting a session with a System DCB 119
GC - Getting Compile unit information 119
GD - Getting Data from any class 120

GE - Getting External Symbol Dictionary data . . . 121
GN - Getting Names of sections or classes . . . 122
RC - Return Code information . . . 123
EN - Ending a session . . . 123
The Unitary interface . . . 123
The IEWBFDA macro . . . 124
Unitary parameter list . . . 127
Error handling . . . 127
Return and reason codes . . . 128

Chapter 5. IEWBNDD, IEWBNDDX - Binder C/C++ API DLL functions 131
Using the binder C/C++ API and headers . . . 132
Environment . . . 134
Binder API functions . . . 135
__iew_addA() – Add alias . . . 135
__iew_alignT() – Align text . . . 135
__iew_alterW() – Alter workmod . . . 136
__iew_autoC() – Perform incremental autocall . . . 137
__iew_bindW() – Bind workmod . . . 138
__iew_closeW() – Close workmod . . . 138
__iew_getC() – Get compile unit list . . . 139
__iew_getD() – Get data . . . 139
__iew_getE() – Get ESD data . . . 140
__iew_getN() – Get names . . . 141
__iew_import() – Import a function or external variable . . . 142
__iew_includeDeptr() – Include module via DEPTR . . . 142
__iew_includeName() – Include module via NAME . . . 143
__iew_includePtr() – Include module via POINTER . . . 144
__iew_includeSmde() – Include module via SMDE . . . 145
__iew_includeToken() – Include module via DEPTR . . . 145
__iew_insertS() – Insert section . . . 146
__iew_loadW() – Load workmod . . . 147
__iew_openW() – Open workmod . . . 147
__iew_orderS() – Order sections . . . 149
__iew_putD() – Put data . . . 149
__iew_rename() – Rename symbolic references . . . 150
__iew_resetW() – Reset workmod . . . 151
__iew_saveW() – Save workmod . . . 151
__iew_setL() – Set library . . . 152
__iew_setO() – Set option . . . 153
__iew_startS() – Start segment . . . 153
Binder API utility functions . . . 154
__iew_create_list() – Create list . . . 154
__iew_eod() – Test for end of data . . . 155
__iew_get_reason_code() – Get a reason code from API context . . . 155
__iew_get_return_code() – Get a return code from API context . . . 156
__iew_get_cursor() – Get cursor value . . . 156
__iew_set_cursor() – Set cursor value . . . 157
Fast data functions . . . 157
__iew_fd_end() – End a session . . . 157
__iew_fd_getC() – Get compile unit list . . . 158
__iew_fd_getD() – Get data . . . 159
__iew_fd_getE() – Get ESD data . . . 159
__iew_fd_getN() – Get names . . . 160
__iew_fd_open() – Open a session . . . 161
__iew_fd_startDcb() – Starting a session with BLDL data . . . 161

__iew_fd_startDcbS() – Starting a session with a System DCB 162
__iew_fd_startName() – Starting a session with a DD name or path 162
__iew_fd_startToken() – Starting a session with a CSVQUERY token . . . 163
Fast data utility functions 163
__iew_fd_eod() – Test for end of data 164
__iew_fd_get_reason_code() – Get a reason code from FD context 164
__iew_fd_get_return_code() – Get a return code from FD context. 164
__iew_fd_get_cursor() – Get cursor value 165
__iew_fd_set_cursor() – Set cursor value. 166
Binder API and fast data API common utility functions 166
__iew_api_name_to_str() – Convert API name into string 166
Binder API return and reason codes 167
Fast data access return and reason codes 167

Chapter 6. Invoking the binder program from another program 169

Chapter 7. Setting options with the regular binder API 173
Setting options with the binder API 173

Chapter 8. User exits. 177
Execution environment 177
Registers at entry to the user exit routine. 177
Message exit 178
Save exit 178
Interface validation exit 180

Appendix A. Object module input conventions and record formats . . . 185
Input conventions 185
Record formats 186
SYM record 186
ESD record. 187
Text record 188
RLD record. 189
END record. 190
Extended object module (XOBJ) 191
XSD record format 192
Notes 193

Appendix B. Load module formats 195
Input conventions 195
Record formats 195

Appendix C. Generalized object file format (GOFF) 203
Guidelines and restrictions 203
Incompatibilities 204
GOFF record formats 204
Conventions 205
Module header record. 207
External symbol definition record. 208
Text record 218
Relocation directory record 222
Deferred element length record 227
Associated data (ADATA) record 228
End of module record 229
Mapping object formats to GOFF format 230
Module header records 231

Mapping object module ESD elements to GOFF format 231
Mapping object module XSD items to GOFF format 234
Mapping object module TXT items to GOFF format 234
Mapping object module RLD items to GOFF format 235
Mapping object module END items to GOFF format 235
Generating DLLs and linkage descriptors from GOFF object modules 237
Exports 237
Imports 237
Using linkage descriptors in non-DLL applications 238

Appendix D. Binder API buffer formats 239
Buffer header contents 241
Version 1 buffer formats 242
ESD entry (version 1) 243
Binder identification data (version 1) 245
Language processor identification data (version 1) 246
User identification data (version 1) 247
AMASPZAP identification data (version 1) 248
RLD entry (version 1) 249
Internal symbol table (version 1) 250
Text data buffer (version 1) 250
Binder name list (version 1) 251
Extent list (version 1) 251
Version 2 buffer formats 252
ESD entry (version 2) 253
RLD entry (version 2) 257
Binder name list (version 2) 259
Module map (version 2) 260
Migration to version 2 buffers 261
Field correspondence for ESD records 262
ESD conversion notes (and PM1-PM2 differences) 263
Field correspondence for RLD records 266
RLD conversion notes (and PM1-PM2 differences) 267
Version 3 buffer formats 267
ESD entry (version 3) 268
RLD entry (version 3) 272
PARTINIT entry (version 3) 274
Migration to version 3 buffers 274
Part initializers 274
ESD conversion notes 274
RLD conversion notes 275
Version 4 buffer formats 275
Binder name list (version 4) 275
Version 5 buffer formats 276
PMAR entry (version 5) 277
ESD entry (version 5) 278
Version 6 buffer formats 281
LIB entry (version 6) 282
CUI entry (version 6) 283
Binder name list (version 6) 285
Version 7 buffer formats 285
Language processor identification data (version 7) 286
Binder name list (version 7) 287

Appendix E. Data areas 289
PDS directory entry format on entry to STOW 289

Cross reference . . . 294
PDS directory entry format returned by BLDL . . . 296
Cross reference . . . 301
PDSE directory entry returned by DESERV (SMDE data area) . . . 303
Accessing program object class information . . . 306
Module Map . . . 308
Structure of the module map . . . 309
Example of record structure . . . 309
Compile unit information . . . 310
Module map format . . . 310

Appendix F. Programming examples for binder APIs . . . 313
Examples for binder regular API . . . 315
Examples for binder C/C++ API . . . 323
Examples for fast data access API . . . 332
Examples of JCL . . . 343

Appendix G. Using the transport utility (IEWTPORT) . . . 347
Executing IEWTPORT . . . 347
Defining the data sets . . . 348
Allocating space for the SYSUTn data sets . . . 348
Transporting selected members . . . 348
Sample IEWTPORT invocations . . . 349
Convert a program object to a transportable program . . . 349
Convert an entire program library . . . 349
Convert a transportable program to a single program object . . . 349
Messages, errors, and return codes . . . 349
Messages and codes . . . 349
Errors . . . 350
Return codes . . . 350
Logical structure of a transportable file . . . 350
Mapping macro IEWTFMT . . . 351
Header . . . 352
Trailer . . . 353
Transportable program . . . 353

Appendix H. Establishing installation defaults . . . 359
Binder APIs . . . 360

Appendix I. Accessibility . . . 361
Using assistive technologies . . . 361
Keyboard navigation of the user interface . . . 361
z/OS information . . . 361

Notices . . . 363
Programming interface information . . . 365
Policy for unsupported hardware . . . 365
Trademarks . . . 365

Index . . . 367

Figures

1. Data items . . . 5
2. IEWBUFF function summary . . . 13
3. Example of a variable length string parameter . . . 21
4. IEWBIND function call summary . . . 25
5. Rename list . . . 45
6. SYM input record . . . 186
7. ESD input record . . . 187
8. Text input record . . . 188
9. RLD input record . . . 189
10. END input record–type 1 . . . 190
11. END input record–type 2 . . . 190
12. IDR data in an object module END record . . . 191
13. SYM record (load module) . . . 195
14. CESD record (load module) . . . 196
15. Scatter/Translation record . . . 197
16. Control record (load module) . . . 198
17. Relocation dictionary record (load module) . . . 199
18. Control and relocation dictionary record (load module) . . . 200
19. Record format of load module IDRs–part 1 . . . 200
20. Record format of load module IDRs–part 2 . . . 201
21. Record format of load module IDRs–part 3 . . . 202
22. Current IDR data item (format 1) . . . 221
23. Extended IDR data item (format 2) . . . 221
24. IDR data item (format 3) . . . 221
25. Format for ESD entries . . . 243
26. Format for binder identification data . . . 245
27. Format for language processor identification data . . . 246
28. Format for user identification data . . . 247
29. Format for AMASPZAP identification data . . . 248
30. Format for RLD entries . . . 249
31. Format for symbol table (SYM) entries . . . 250
32. Format for TXT entries . . . 250
33. Format for binder name list entries . . . 251
34. Format for contents extent list entries . . . 251
35. Format for ESD entries . . . 254
36. Format for RLD entries . . . 257
37. Format for binder name list entries . . . 259
38. Format for module map list entries . . . 260
39. Format for ESD entries . . . 269
40. Format for RLD entries . . . 272
41. Format for PARTINIT entries . . . 274
42. Format for binder name list entries . . . 276
43. Format for PMAR entries . . . 277
44. Format for ESD entries . . . 278
45. Format for LIB entries . . . 282
46. Format for CUI entries . . . 283
47. Format for binder name list entries . . . 285
48. Format for IDRL entries . . . 286
49. Format for binder name list entries . . . 288
50. Transportable file structure . . . 351
51. Transportable program structure . . . 353
52. Transportable program body structure . . . 354
53. Alias data record . . . 354

© Copyright IBM Corp. 1991, 2012

54. Attributes Data Record 355
55. Item data record 357

Tables

1. Section names 4
2. Processing intent and calls 10
3. Common binder API reason codes 22
4. ADDA parameter list 30
5. ALIGNT parameter list 32
6. ALTERW parameter list 35
7. AUTOCall parameter list 37
8. BINDW parameter list 41
9. CREATEW parameter list 42
10. DELETEW parameter list 43
11. DLLRename parameter list 46
12. ENDD parameter list 47
13. GETC parameter list 49
14. GETD parameter list 52
15. GETE parameter list 56
16. GETN parameter list 58
17. IMPORT parameter list 60
18. INCLUDE parameter list 67
19. INSERTS parameter list 69
20. LOADW parameter list 71
21. ORDERS parameter list 73
22. PUTD parameter list 77
23. RENAME parameter list 79
24. RESETW parameter list 81
25. SAVEW parameter list 86
26. SETL parameter list 88
27. SETO parameter list 91
28. Binder list structure 93
29. SYSLIB data set DCB parameters 94
30. SYSPRINT DCB parameters 95
31. SYSTERM DCB parameters 95
32. STARTD parameter list 97
33. STARTS parameter list 99
34. All binder API reason codes 99
35. SB parameter list 117
36. SJ parameter list 118
37. SQ parameter list 118
38. SS parameter list 119
39. GC parameter list 120
40. GD parameter list 121
41. GE parameter list 122
42. GN parameter list 122
43. RC parameter list 123
44. EN parameter list 123
45. IEWBFDA parameter list 127
46. Accessing binder C/C++ API through DLL support 134
47. Setting options with the binder API 173
48. GOFF record-type identification prefix byte 206
49. Module header record 207
50. External symbol definition record 208
51. External symbol definition continuation record 211
52. Relationship of an element's ESDIDs and parent ESDIDs 211
53. Specifiable external symbol definition record items 211

© Copyright IBM Corp. 1991, 2012

54. External symbol definition behavioral attributes . . . 212
55. Specifiable external symbol behavioral attributes. . . . 216
56. Extended attribute information data structures. . . . 217
57. Data element descriptor data structures . . . 217
58. Supported extended attribute type codes . . . 218
59. Text record . . . 218
60. Text continuation record . . . 220
61. Identification record data field. . . . 220
62. Text encoding types . . . 222
63. Relocation directory record. . . . 222
64. Relocation directory data element . . . 223
65. Relocation directory continuation record . . . 223
66. Relocation directory data element flags field . . . 224
67. RLD-element referent and reference types . . . 227
68. Deferred section-length record . . . 227
69. Deferred element-length data item . . . 228
70. Associated data (ADATA) record type assignments . . . 228
71. End-of-module record, with optional entry point request . . . 229
72. End-of-module (entry point name) continuation record. . . . 230
73. Mapping OBJ ESD SD items to GOFF format. . . . 232
74. OBJ treatment of ESD PC items and blank names . . . 232
75. Mapping OBJ ESD LD items to GOFF format . . . 233
76. Mapping OBJ ESD ER/WX items to GOFF format . . . 233
77. Mapping OBJ ESD PR items to GOFF format. . . . 234
78. Mapping OBJ TXT items to GOFF format . . . 235
79. Mapping OBJ RLD items to GOFF format . . . 235
80. Mapping OBJ END IDR items to GOFF format . . . 236
81. Mapping OBJ END section–length items to GOFF format . . . 236
82. Mapping OBJ END–entry items to GOFF format . . . 236
83. XPLINK 'call-by'name' descriptor adcon offsets . . . 238
84. API buffer formats . . . 240
85. API buffer definitions . . . 240
86. Buffer header format . . . 241
87. Buffer type format . . . 242
88. Comparison of new and old ESD formats . . . 262
89. ESD element usage . . . 265
90. Comparison of new and old RLD formats . . . 266
91. SMDE format. . . . 303
92. Directory entry name section . . . 305
93. Directory entry notelist section (PDS only) . . . 305
94. Directory entry token section (normal use) . . . 305
95. Directory entry token section (system DCB) . . . 305
96. z/OS UNIX System Services file descriptor section . . . 305
97. Directory entry primary name section . . . 305
98. BLIT structure format . . . 306
99. BLIT class entry format . . . 307
100. Module map header . . . 310
101. Map entry format . . . 310
102. Compile unit information. . . . 311
103. IEWTPORT return codes . . . 350
104. Transportable program record . . . 352
105. Transportable file header . . . 352
106. Transportable file trailer . . . 353
107. Transportable program descriptor map . . . 353
108. Transportable program alias data header . . . 355
109. Transportable program attributes data header . . . 356

110. Transportable program item data header 357

About this document

This book is intended to help you learn about and use programming interfaces to create, modify, or obtain information about program objects and load modules. It contains general-use programming interface and associated guidance information.

The interfaces described here are provided by the program management component of z/OS, including the program management binder, the linkage editor, the batch loader, and the transport utility.

- Chapter 1 contains background information as well as a general introduction to the binder.
- Chapter 2 contains information on the binder API interface macros.
- Chapter 3 contains information on all aspects of the IEWBIND function.
- Chapter 4 explains how to more efficiently obtain module data from a program object with fast data access service.
- Chapter 5 explains how to set up binder options for advanced facilities using the binder API.
- Chapter 6 explains how to invoke the binder program from another program.
- Chapter 7 explains how to specify options with the binder API.
- Chapter 8 explains how to define user exits.
- Appendix A describes object module input and record formats.
- Appendix B describes linkage editor load module formats.
- Appendix C describes the conventions and formats for the generalized object file format (GOFF).
- Appendix D describes binder API buffer formats.
- Appendix E describes IHAPDS data areas.
- Appendix F describes programming examples for the binder API.
- Appendix G explains how to use the transport utility.
- Appendix H explains how to establish installation default values for binder options.
- Appendix I explains how to help users with physical disabilities to use software products successfully.

Required product knowledge

To use this book effectively, you should be familiar with the following:

- A thorough understanding of program binder concepts, options, and control statements. See *z/OS MVS Program Management: User's Guide and Reference*.
- Familiarity with the types of parameter lists and data structures used in z/OS programming.
- Some knowledge of z/OS assembler terminology.

Required publications

You should be familiar with the information presented in following publications:

Publication Title	Order Number
z/OS MVS Program Management: User's Guide and Reference	SA22-7463
HLASM Language Reference	SC26-4940

© Copyright IBM Corp. 1991, 2012

Related publications

The following publications might be helpful:

Publication Title	Order Number
z/OS XL C/C++ Programming Guide	SC09-4765
z/OS XL C/C++ User's Guide	SC09-4767
z/OS MVS Diagnosis: Reference	GA22-7588
z/OS MVS Diagnosis: Tools and Service Aids	GA22-7589
z/OS MVS JCL User's Guide	SA22-7598
z/OS MVS System Messages, Vol 8 (IEF-IGD)	SA22-7638
z/OS UNIX System Services Command Reference	SA22-7802
HLASM Programmer's Guide	SC26-4941

Referenced publications

Within the text, references are made to other z/OS books and books for related products. The titles and order numbers are listed in the following table:

Publication Title	Order Number
z/OS MVS Programming: Assembler Services Guide	SA22-7605
z/OS MVS System Messages, Vol 8 (IEF-IGD)	SA22-7638

Notational conventions

A uniform notation describes the syntax of the control statements documented in this publication. This notation is not part of the language; it is merely a way of describing the syntax of the statements. The statement syntax definitions in this book use the following conventions:

[ ] Brackets enclose an optional entry. You can, but need not, include the entry. Examples are:
[*length***]**
[MF=E]

| An OR sign (a vertical bar) separates alternative entries. You must specify one, and only one, of the entries unless you allow an indicated default. Examples are:
[REREAD | LEAVE]
[*length***|'S']**

{ } Braces enclose alternative entries. You must use one, and only one, of the entries. Examples are:
BFTEK={S | A}
{K | D}
{*address* **| S | O}**

Sometimes alternative entries are shown in a vertical stack of braces. An example is:

```
MACRF={{(R[C|P])}
       {(W[C|P|L])}
       {(R[C],W[C])}}
```

In the preceding example, you must choose only one entry from the vertical stack.

. . . An ellipsis indicates that the entry immediately preceding the ellipsis can be repeated. For example:

(*dcbaddr***,[(***options***)],. . .)**

‘ ’ A ‘ ’ indicates that a blank (an empty space) must be present before the next parameter.

UPPERCASE BOLDFACE

Uppercase boldface type indicates entries that you must code exactly as shown. These entries consist of keywords and the following punctuation symbols: commas, parentheses, and equal signs. Examples are:

CLOSE , , , ,TYPE=T
MACRF=(PL,PTC)

UNDERSCORED UPPERCASE BOLDFACE

Underscored uppercase boldface type indicates the default used if you do not specify any of the alternatives. Examples are:

[EROPT={ACC | SKP | ABE}]
[BFALN={F | D}]

Lowercase italic

Lowercase italic type indicates a value to be supplied by you, the user, usually according to specifications and limits described for each parameter. Examples are:

number
image-id
count

REQUIRED KEYWORDS AND SYMBOLS

Entries shown **IN THE FORMAT SHOWN HERE** (notice the type of highlighting just used) must be coded exactly as shown. These entries consist of keywords and the following punctuation symbols: commas, parentheses, and equal signs. Examples are:

CLOSE , , , ,TYPE=T
MACRF=(PL,PTC)

Note: The format (the type of highlighting) used to identify this type of entry depends on the display device used to view a softcopy book. The published hardcopy version of this book displays this type of entry in uppercase boldface type.

DEFAULT VALUES

Values shown **IN THE FORMAT SHOWN HERE** (notice the type of highlighting just used) indicate the default used if you do not specify any of the alternatives. Examples are:

[EROPT={ACC | SKP | ABE}]
[BFALN={F | D}]

Note: The format (the type of highlighting) used to identify this type of entry depends on the display device used to view a softcopy book. The published hardcopy version of this book displays this type of entry in underscored uppercase boldface type.

User specified value

Values shown *in the format shown here* (notice the type of highlighting just used) indicate a value to be supplied by you, the user, usually according to specifications and limits described for each parameter. Examples are:

number
image-id
count

Where to find more information

Where necessary, this document references information in other documents, using shortened versions of the document title. For complete titles and order numbers of the documents for all products that are part of z/OS, see *z/OS Information Roadmap*.

You might also need the following information:

Short Title Used in This Document	Title	Order Number
SNA Sync Point Services Architecture	*Systems Network Architecture Sync Point Services Architecture Reference*	SC31-8134

Information updates on the web

For the latest information updates that have been provided in PTF cover letters and Documentation APARs for z/OS®, see the online document at http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/Shelves/ODOCAPAR

This document is updated weekly and lists documentation changes before they are incorporated into z/OS publications.

The z/OS Basic Skills Information Center

The z/OS Basic Skills Information Center is a Web-based information resource intended to help users learn the basic concepts of z/OS, the operating system that runs most of the IBM mainframe computers in use today. The Information Center is designed to introduce a new generation of Information Technology professionals to basic concepts and help them prepare for a career as a z/OS professional, such as a z/OS system programmer.

Specifically, the z/OS Basic Skills Information Center is intended to achieve the following objectives:

- Provide basic education and information about z/OS without charge
- Shorten the time it takes for people to become productive on the mainframe
- Make it easier for new people to learn z/OS.

To access the z/OS Basic Skills Information Center, open your Web browser to the following Web site, which is available to all users (no login required): http://publib.boulder.ibm.com/infocenter/zos/basics/index.jsp

How to send your comments to IBM

We appreciate your input on this publication. Feel free to comment on the clarity, accuracy, and completeness of the information or give us any other feedback that you might have.

Use one of the following methods to send us your comments:

1. Send an email to mhvrcfs@us.ibm.com
2. Visit the Contact z/OS web page at http://www.ibm.com/systems/z/os/zos/webqs.html
3. Mail the comments to the following address:
   IBM Corporation
   Attention: MHVRCFS Reader Comments
   Department H6MA, Building 707
   2455 South Road
   Poughkeepsie, NY 12601-5400
   U.S.A.
4. Fax the comments to us as follows:
   From the United States and Canada: 1+845+432-9405
   From all other countries: Your international access code +1+845+432-9405

Include the following information:

- Your name and address
- Your email address
- Your telephone or fax number
- The publication title and order number:
  z/OS V1R13.0 MVS Program Management: Advanced Facilities
  SA22-7644-14
- The topic and page number related to your comment
- The text of your comment.

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

IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you submit.

If you have a technical problem

Do not use the feedback methods listed above. Instead, do one of the following:

- Contact your IBM service representative
- Call IBM technical support
- Visit the IBM support portal at http://www.ibm.com/systems/z/support/

© Copyright IBM Corp. 1991, 2012

Summary of changes

This document contains terminology, maintenance, and editorial changes to improve consistency and retrievability. Technical changes or additions to the text and illustrations are indicated by a vertical line to the left of the change.

Changes made in z/OS Version 1 Release 13, as updated September 2012

The book contains information previously presented in *z/OS MVS Program Management: Advanced Facilities*, SA22-7644-13, which supports z/OS Version 1 Release 13.

New information:

- A new VERSION value for many Binder API functions was added, see "IEWBIND function reference" on page 24.

Changed information:

- Information in Chapter 5, "IEWBNDD, IEWBNDDX - Binder C/C++ API DLL functions," on page 131 was clarified.

Changes made in z/OS Version 1 Release 13

The book contains information previously presented in *z/OS MVS Program Management: Advanced Facilities*, SA22-7644-12, which supports z/OS Version 1 Release 12.

New information:

- A new return and reason code has been added for SAVEW: Save workmod. See "Return and reason codes" on page 83.
- New PRIV parameter has been added to the STRIPSEC option. See Table 47 on page 173.
- A new field (Conditional Sequential Resolution Flag) has been defined in GOFF architecture. See Table 66 on page 224.
- Updated RLD_BIND_ATTR field in RLD entry (version 3) to support the new conditional sequential resolution bit. See "RLD entry (version 3)" on page 272.

Changes made in z/OS Version 1 Release 12

The book contains information previously presented in *z/OS MVS Program Management: Advanced Facilities*, SA22-7644-11, which supports z/OS Version 1 Release 11.

New information:

- Both the binder and fast data access service support the following dynamic allocation (DYNALLOC or SVC 99) options for all data sets: S99TIOEX(XTIOT), S99ACUCB (NOCAPTURE), and S99DSABA (DSAB above the line). See "INCLUDE: Include module" on page 61 and "Parameter descriptions" on page 110.
- Fast data access service supports accessing PMAR data and a program loaded in z/OS UNIX System Services files. See "Parameter descriptions" on page 110 in Chapter 4, "IEWBFDAT - Binder Fast data access API functions," on page 109.

© Copyright IBM Corp. 1991, 2012

- XPLINK version of C front end can improve performance for XPLINK callers of Binder's C/C++ API. See “Using the binder C/C++ API and headers” on page 132.
- The module map IEWBMMP is added in the C header file. See “Using the binder C/C++ API and headers” on page 132.
- New programming examples are added. See Appendix F, “Programming examples for binder APIs,” on page 313.
- Both the binder and fast data access service support all data sets allocated in the extended addressing space (EAS) of an extended address volume (EAV). See “INCLUDE: Include module” on page 61 and “Using the fast data access service” on page 109.

Changed information:

- The "Readers' Comments - We'd Like to Hear from You" section at the back of this publication has been replaced with a new section “How to send your comments to IBM” on page xix. The hardcopy mail-in form has been replaced with a page that provides information appropriate for submitting readers comments to IBM.

Changes made in z/OS Version 1 Release 11

The book contains information previously presented in *z/OS MVS Program Management: Advanced Facilities*, SA22-7644-10, which supports z/OS Version 1 Release 10.

New information:

- A new Binder DD statement IEWPARMS is added to support an alternate source for binder options. See Table 47 on page 173.
- New processing note about SIGN option in the INCLUDE module section. See “INCLUDE: Include module” on page 61.

Changed information:

- The definition of VERSION value for Binder regular API functions is updated in “IEWBIND function reference” on page 24.
- Return codes of the following IEWBIND functions have been changed in “IEWBIND function reference” on page 24:
  - BINDW: Bind workmod
  - DELETEW: Delete workmod
  - RESETW: Reset workmod
  - SAVEW: Save workmod

Chapter 1. Using the binder application programming interfaces (APIs)

This topic describes how an application program, system service, or utility can request binder services under program control. Your program can use the binder application programming interface to invoke individual services such as Include, Set Option, Get Data, and Get ESD. By combining calls to these services in a logical order, your program can perform operations on program modules.

The binder also provides two sets of higher level interfaces which perform all operations in a single call, but without the same level of control or function. The IEWBFDA service described in Chapter 4, “IEWBFDAT - Binder Fast data access API functions,” on page 109 provides a very efficient way to extract data from an existing program object. Invoking the binder from a program (169) is similar to executing the binder as a batch program, but with your program controlling options and DD names.

Much of the material in this chapter is useful regardless of the interface you choose to use. The material in the two chapters which follow this one is specific to the low level interfaces.

The low level binder application programming interface consists of assembler macros and user exit points. The IEWBIND macro provides the invoking program with access to the binder services. The IEWBUFF macro provides maps of data areas passed as parameters on the IEWBIND macro invocation. The message user exit allows you to suppress printing of a message. The save user exit allows you to retry a failed attempt to store a member or alias name. The interface validation exit allows your exit routine to examine descriptive data for both caller and called at each external reference.

The binder can be called from programs written in assembler and in other languages. If your program is written in a language other than assembler, you will need to encode the parameter lists for each binder call. See “Invoking the binder without the macros” on page 20 for tips on how to invoke the binder from non–assembler language programs.

Understanding binder programming concepts

Before you begin designing programs for the binder application programming interface, you should understand several of its programming concepts.

dialog

A programmed session with the binder is called a *binder dialog*. You begin the session with a request to start a dialog. This request establishes the environment for binder processing during that dialog, including obtaining work space, initializing control blocks, and specifying DD names for the standard data sets to be used.

Your program can start more than one dialog and maintain more than one dialog at the same time. The dialogs are independent. You can perform operations on more than one dialog concurrently, but you cannot pass binder data directly from one dialog to another.

When any dialog is ended, all buffers and control information relating to that dialog are deleted.

© Copyright IBM Corp. 1991, 2012

dialog token
Each dialog is identified by a unique *dialog token*. The dialog token is created when you request a new dialog. You use the token corresponding to a specific dialog when you request services, so the binder can perform the services on the correct input and return the desired results to the correct dialog. The token is invalidated when the dialog is ended.

workmod
An area of working storage used to create or operate on a program module is called a *workmod*. After a dialog has been established, your program issues requests to create a workmod, to reset a workmod to the null state, or to delete a workmod. At least one workmod must be created before any module operations can be requested.

A dialog can have many workmods associated with it, but each workmod is associated with only one dialog. Binder-generated data cannot be passed between workmods. Ending a dialog causes any remaining workmods associated with it to be deleted.

workmod token
Each workmod is identified by a unique *workmod token*. The workmod token is created when the request for a new workmod is made. The workmod token associates the workmod with a particular dialog. Your program must pass the token as a parameter on all requests involving that workmod. The token is invalidated when either the workmod is deleted or the dialog is ended.

processing intent
Processing intent specifies the range of binder services that may be requested for a workmod. The processing intent is set to either *bind* or *access* when the workmod is created. Bind allows editing of the modules; access does not. See “Processing intents” on page 10 for detailed information.

element
An *element* is a named portion of module data in a workmod that is directly addressable by the binder. Program modules logically consist of elements. Each element is identified by its data class identifier and, optionally, a section name.

class A *class* identifier is a required descriptor for each element. It identifies the type and the format of the data element and determines the operations that can be performed on it. The following binder-defined classes are valid:

B_TEXT
Relocatable text, including executable machine instructions, constants, and, in the case of the nonreentrant programs, data areas defined within the program. See also B_TEXT24 and B_TEXT31.

B_PRV
Pseudoregister vector (a data area shared across compile units) that is used by high-level languages and assembler code using DXD and CXD instructions.

B_ESD
External Symbol Dictionary, a catalog containing all symbols that are available to the binder. These symbols include section and class names, as well as ordinary symbol definitions and references.

B_RLD
Relocation dictionary, containing information used to adjust addresses within the code based on where the module is loaded.

B_IDRB
Binder identification record, indicating the binder version, size, and how and when the program object was created.

B_IDRL
Language processor identification record, indicating the compilers that created the binder input, the date of compilation, and the binder size.

B_IDRU
User-specified identification record, containing data provided on an IDENTIFY control statement.

B_IDRZ
AMASPZAP identification record, for programs that have been modified at the object level.

B_SYM
Internal symbol table records

B_MAP
Module map

B_TEXT24
Relocatable text loaded below the 16-meg line and used in place of B_TEXT if RMODE=SPLIT is specified.

B_TEXT31
Relocatable text loaded above the 16-meg line and used in place of B_TEXT if RMODE=SPLIT was specified.

B_LIT Load information table, containing data generated by the binder and used by the Language Environment® run time. For more information, see “Accessing program object class information” on page 306.

B_IMPEXP
IMPORT/EXPORT table, containing data generated by the binder and used by the Language Environment run time to support DLLs and DLL applications.

B_PARTINIT
Part initializers, containing the initial data for parts in merge classes. During bind processing, merge classes are mapped at the module level and the initial data moved from B_PARTINIT to the owning merge class.

B_DESCR
Linkage descriptors built by the binder for XPLINK support (Linkage Descriptors may also exist in compiler-defined classes).

In addition to the binder defined classes shown above, compiler-defined class names and class names that are defined by you (or your system programmer) can be specified on all calls. Class names are limited to 16 characters. Class names beginning with C_ should be used only for Language Environment-enabled applications.

section
A *section* name is a programmer-, assembler-, or compiler-assigned name

of a collection of one or more elements, where each element belongs to a specified class. All binder actions referring to sections affect all the elements in that section. For control and common sections, the section name is the CSECT or common section name.

Some elements do not relate directly to any particular section (for example, the binder identification records). The binder assigns these elements unique section names.

Unnamed sections (sections with names consisting of all blanks), and sections whose names the user has designated as not visible to the other sections in the module ('section scope'), are given unique binder–generated names. Binder–generated names are fullword integers. They are printed in binder listings as $PRIV*xxxxxx*, where *xxxxxx* is the printable representation of the integer. The one exception is 'blank common' whose name is retained as a one-character blank.

Section names may be assigned by the user or by the binder. For each of the binder-defined classes just described, the sources of the definitions for the section names are shown in the following table:

Table 1. Section names

Class Name	Section Name
B_TEXT	User specified
B_PRV	User specified; binder retains them in binder-created section X'00000003'
B_ESD	User specified
B_RLD	User specified
B_IDRB	Binder-created class name, in binder-created section X'00000001'
B_IDRL	User specified
B_IDRU	User specified
B_IDRZ	User specified
B_SYM	User specified
B_MAP	Binder-created class name, in binder-created section X'00000001'
B_TEXT24	User specified or binder-created
B_TEXT31	User specified or binder-created
B_LIT	Binder-created class name, in binder-created section IEWBLIT
B_IMPEXP	Binder-created class name, in binder-created section IEWBCIE
B_PARTINIT	User specified
B_DESCR	Binder-created class name, in binder-created section X'00000003'

A program module can be visualized as a two-dimensional grid with the class type on one axis and the section name on the other, as shown in Figure 1 on page 5. Any class type can appear in more than one section, and any section can contain elements in multiple classes. Some elements relate to the entire module, not to any

particular section, and do not have a section name.

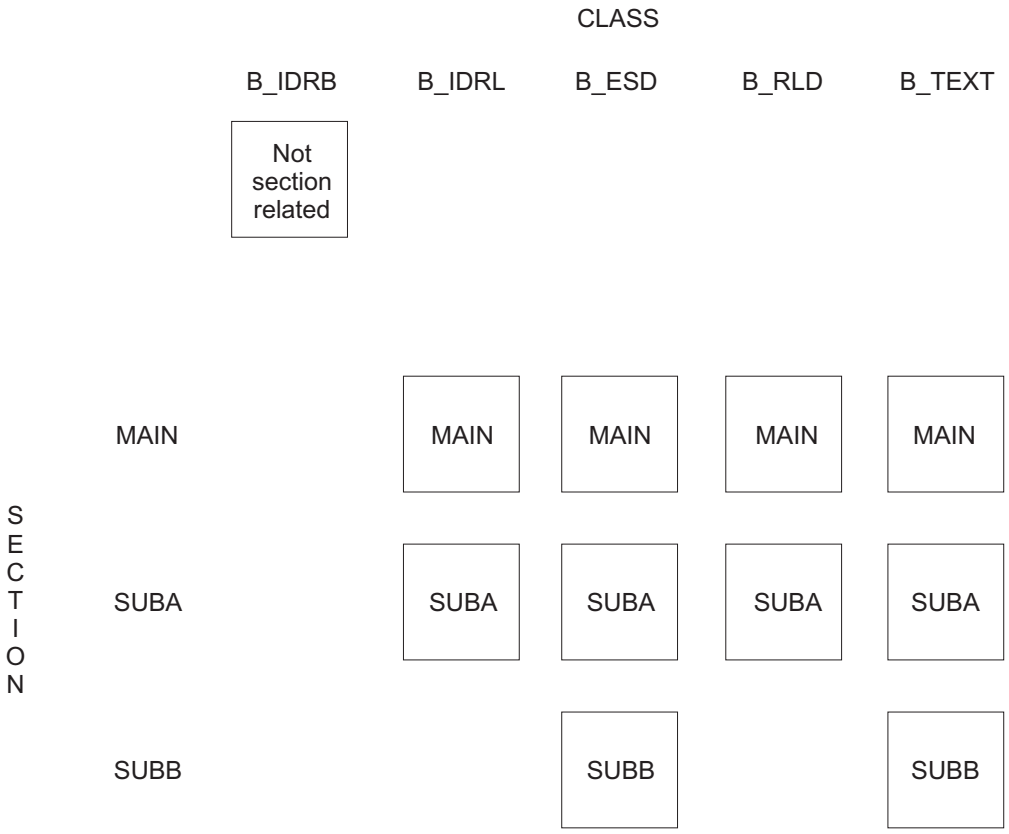


Figure 1. Data items. The workmod shown in this example contains three sections, MAIN, SUBA, and SUBB. The B_IDRB item shown does not relate to a particular section. Note that items do not need to exist for all class and section combinations.

Symbol names in API calls

The abbreviated names used in binder and AMBLIST listings are solely for readability purposes. The full (non-abbreviated) name must be used in binder API calls in section or symbol names in the API parameter lists.

The names which appear in binder and AMBLIST output listings as $PRIVxxxxxx, where xxxxxx is a hexadecimal number, are printable representations of binder-generated names. The actual names are fullword binary integers (for which xxxxxx is the hexadecimal equivalent). The fullword integer must be used in binder API calls.

Similarly, symbol names in GETD, GETE, or GETN buffers are always the true internal names (not abbreviated, and fullword integers if they are generated names).

Program management support of class attributes

Starting with version 2, program objects support multiple text classes. Each class has a name and a set of class attributes. Not all class attributes defined for GOFF format object modules and program objects are supported by other z/OS operating system components.

Classes may be combined into segments (based on class attributes) by the binder. Each segment will be loaded into contiguous storage by the program management loader. The criteria used by the binder to combine classes into segments may

change from release to release. The only criteria currently used for dividing a program into segments are class loading behavior (initial/defer) and rmode.

The following section discusses the current level of binder support for the more important class attributes defined for the ED record in GOFF object modules:

alignment
The binder supports doubleword, quadword and page alignments for classes. A class is assigned the most restrictive alignment of any element or part comprising the class, but at least doubleword. The loader supports doubleword and page alignment. A segment containing any classes which are quadword–aligned are marked as page–aligned.

rmode
The binder supports rmodes 24, ANY, or 64. If a save is done to a PDS load module, all rmode 64 fields in ESDs are changed to rmode ANY. rmode 64 is treated as rmode ANY by the LOADW API call and the batch loader interfaces. If a save is done to a program object, the ESDs remain as rmode 64 in the saved module, but the loader honors rmode 64 only for a deferred load class.

class loading attributes
Do not load with module and **Deferred** are fully supported and are used when combining classes into segments. **Movable**,**Shareable** and **Read only** attributes are stored in the saved program object but do not currently have any effect on processing.

descriptive or text
Descriptive classes are never loaded with the module and cannot contain adcons. Examples are B_ESD or user-defined ADATA classes.

binding method
Catenate classes are ordinary text classes. **Merge** classes are composed of parts defined by PR (part reference) records and are mapped into a single element at the module level. For example, PR records of the same name from different input sections are mapped to a single module-level PD (part definition)

data or code (executable or non-executable)
This attribute is saved in the program object but has no effect on processing.

namespace
The **namespace** of a class must be the same as the **namespace** of all symbols (LD/ER/PR ESD names) in that class. **namespace 1** is used for all non-merge classes. **namespace 2** and **namespaces 3** are valid only for merge classes. **namespace 2** is used for pseudoregisters and **namespace 3** is used for parts. See Chapter 2 in *z/OS MVS Program Management: User's Guide and Reference* for more information.

fill Defines a fill byte used for all areas of the class that have no initializing data.

usability attributes
Usability attributes in ED records (RENT, REUS, and so on) are not supported by the binder

Organization of data in the program module

The following information about the location and order of the data in a program module is provided to assist in the design of programs using the GETE or GETD API calls to extract data from a program module, or PUTD API calls to add data.

Module level information

Data which does not pertain to a particular section is kept in *module level sections* which have binder-generated names.

Section 1 (X'0000001') contains the sole elements of classes B_IDRB and B_MAP. It also contains PR and ER ESD records for symbols which are not the target of any adcon.

Section 3 (X'0000003') contains the ESD items (parts) which map the MRG text classes. Examples are B_PRV (the PL/I pseudoregister vector table) and C_WSA (writable static which is used for data items in reentrant C or C++ programs). Although the parts comprising a MRG class may come in from different input sections, the entire class will be mapped as a single element under the appropriate class name in section 3. In addition to the ESDs, section 3 contains any initialization data for the parts in these classes.

Multiple-text class modules, load segments and ESD offsets

In a multi-text class module (supported in PO2 and later program object formats) the binder collects all the text for each class into a contiguous area. If a section has text elements in more than one class, these elements will each be placed in the appropriate class. Thus the text for a given section might no longer reside in a single contiguous area, and *section offset* is no longer a meaningful term. Therefore, for version 2 and higher buffers, offset information can no longer be obtained from an SD ESD record.

Each section contains ED records which define the characteristics of the text classes in that section. Since a section name and class name together define an element, these ED records can also be thought of as describing the attributes of an element. ED records are generated by the binder for load module input, and for traditional and XOBJ object modules. They are generated by the language translator for GOFF format input. The ESD_CLASS_OFFSET field in version 2 and greater buffers contains the offset of the element from the start of its containing class, and the ESD_LENG fields contains the length of the element. This information replaces the CSECT length and offset information in the SD record of a traditional one-dimensional program module.

For object module and load module input, the binder will generate an LD ESD record for each CSECT, with the same name as the CSECT, in class B_TEXT. The offset in the LD record will be the same as the offset in the B_TEXT ED record for the section. For XOBJ format input, the generated LD will be in class C_CODE.

Furthermore, the binder may collect text classes with compatible attributes into load segments. A given load segment will be loaded into contiguous virtual storage. The offset field (ESD_CLASS_OFFSET) in an ESD record in an ESD buffer is the offset from the start of the containing class, not the offset from the start of the segment. For a load segment which contains more than one class, you may want the segment offset rather than the class offset (for example if your intent is to produce a map of the module in virtual storage you will create an incorrect map if you use the class offset).

To compute the segment offset, you should first use the GETN API call to obtain the segment ID and the starting offset of each class from the start of the containing segment. For each ESD record, ESD_RES_CLASS will have the name of the class containing the symbol (the resident class). Then the segment offset can be computed by adding ESD_CLASS_OFFSET to the starting offset of the class within the segment. Segment 1 will always be the segment containing the entry point (the primary and any secondary entry points). In many cases it will be the only segment.

Version number in an API call

The version of an API call determines the format of the parameter list. Although many of the parameter lists are the same for all versions, the formats documented here are specifically for version 6. For earlier formats, see the notes under the description of the VERSION parameter and the Parameter list subtopic for each call. Earlier formats may support a shorter version of the parameter list. The default API call version is VERSION=1.

The version in the buffer header determines the format of the data in the buffer. The default buffer version is VERSION=1. For new programs, we recommend that VERSION=7 be used.

There is no requirement that the buffer version match the program object version. However, you may lose data if the target program has content not supported by the data buffer format you are using.

- Version 2 (and higher) data buffers support all possible contents of a PO2 format program object.
- Version 4 (and higher) data buffers support all possible contents of a PO3 format program object.
- Version 6 (and higher) data buffers support all currently defined contents (as of z/OS 1.3) of a PO4 format program object.
- Version 7 (and higher) data buffers support all currently defined contents (as of z/OS 1.10) of a PO5 format program object.

Binder dialogs

Your program conducts a dialog with the binder using either the IEWBIND and IEWBUFF assembler macros or the equivalent definitions and statements for other programming languages.

Assembler coding summaries of the IEWBIND and IEWBUFF macros are provided in Figure 4 on page 25 and Figure 2 on page 13. Certain function calls are required to establish and terminate the dialog, workmod, and buffers. You select among the remaining calls according to the needs of your program. The following sequence of function calls is recommended:

1. Use STARTD to begin the dialog and to establish a processing environment. A dialog token is returned.
2. Use CREATEW to create a workmod and associate it with the dialog. You pass the dialog token received from the STARTD call and receive a workmod token in return. You specify processing intent on CREATEW as either ACCESS or BIND.
3. Use SETO for setting options and module attributes. You pass either the dialog token or the workmod token. If a dialog token is passed, the options and attributes become the defaults for all workmods attached to that dialog. If a workmod token is specified, any specified options apply only to that workmod.

4. Use INCLUDE to bring one or more modules into the workmod. All control statements and object and program modules are brought into a workmod via the INCLUDE call. If INTENT=ACCESS, one and only one program module can be included in the workmod at a time. Before including another program module, the workmod must be reset.
5. Use ALTERW to make selective changes to symbols or sections in modules already in the workmod or yet to be included. By choosing either the IMMED or NEXT mode of operation, you indicate whether the alteration is applied globally to all modules in the workmod or only to the next module included into the workmod.
6. Several of the function calls do not take place immediately, but are deferred until later in the processing cycle. ALIGNT, INSERTS, ORDERS, and STARTS affect the overall structure of the module and are deferred until BINDW. ADDA provides for specification of aliases and is deferred until SAVEW. You can request these functions in any order, but they must be received prior to the BINDW call.
7. Use the GETC or GETN function calls to get a list of sections or classes for use in subsequent calls. Use the GETD or GETE function calls to extract data from a program module, or use the PUTD function call to add data to it. Before you can use these function calls, your program must define and allocate buffers. Use the GETBUF and INITBUF functions of the IEWBUFF macro to allocate and initialize storage areas to contain the data. Use the MAPBUF function to define DSECT mappings of the areas.
8. Use the BINDW function to instruct the binder to resolve all external references and calculate relocatable address constants. Automatic library call resolves any unresolved external references. Text alignment, ordering, and overlay segmentation are all done at this time.

   Use SETL to specify special call libraries for resolving external references or to indicate that certain references are not to be resolved. SETL requests must precede BINDW.

   If INTENT=BIND, you may issue the BINDW request before any LOADW or SAVEW for a module. LOADW and SAVEW automatically perform bind processing if you did not issue a separate BINDW call previously. If INTENT=ACCESS, a BINDW request is rejected. Binding is required if more than one module has been included, or if other changes have taken place that would affect the size or structure of the module, such as reordering sections.

   Once binding has taken place, no further modifications to the workmod are allowed.
9. Use LOADW or SAVEW to dispose of the module in workmod. LOADW requests that the module be loaded into virtual storage in a format suitable for execution. SAVEW requests that the module be saved in a data set. You can issue both requests for a module in either order.
10. Use DELETEW to delete the workmod and free all associated resources. Use RESETW to free all of the module data in the designated workmod, but to retain the workmod structure. RESETW is functionally equivalent to issuing DELETEW and CREATEW requests in sequence.

    Delete and reset requests are rejected if the workmod is in an altered state. You can force the deletion of an altered workmod by specifying PROTECT=NO on the DELETEW request.
11. When your program is finished processing all of the data obtained from the workmod, use the IEWBUFF FREEBUF function to free the storage for the data.

12. Use ENDD to end the dialog and to free all remaining resources. An ENDD request is rejected if any modified workmods remain, but can be forced by specifying PROTECT=NO.

See Appendix F, “Programming examples for binder APIs,” on page 313 for a typical assembler language application of the binder APIs.

Processing intents

Each time you create or reset a workmod, you associate it with a processing intent. The intent determines the services and options that are valid for the workmod.

CREATEW and RESETW require you to specify the processing intent using the INTENT parameter. The two values are:

BIND The workmod is bound before being saved or loaded. All binder functions can be requested.

ACCESS
The workmod is not bound before being saved or loaded. No services that might alter the size or structure of the program module can be requested.

Table 2 lists each call and indicates whether it is required, optional, or not allowed within a workmod of either processing intent.

Table 2. Processing intent and calls

FUNCTION	INTENT=BIND	INTENT=ACCESS
ADDA	Optional	Optional
ALIGNT	Optional	Not allowed
ALTERW	Optional	Not allowed
AUTOCALL	Optional	Not allowed
BINDW	Required	Not allowed
CREATEW	Required	Required
DELETEW	Optional	Optional
DLLR	Optional	Optional
ENDD	Required	Required
GETC	Optional	Optional
GETD (see note 1)	Optional	Optional
GETE (see note 2)	Optional	Optional
GETN (see note 3)	Optional	Optional
IMPORT	Optional	Not allowed
INCLUDE (see note 4)	Required	Required
INSERTS	Optional	Not allowed
LOADW	Optional	Optional
ORDERS	Optional	Not allowed
RENAME	Optional	Not allowed
RESETW	Optional	Optional
SAVEW	Optional	Optional
SETL	Optional	Not allowed

Table 2. Processing intent and calls (continued)

FUNCTION	INTENT=BIND	INTENT=ACCESS
SETO (see note 5)	Optional	Optional
STARTD	Required	Required
STARTS	Optional	Not allowed

Notes:

1. If INTENT=BIND, the module must be bound before GETD calls can be executed.
2. If INTENT=BIND, the module must be bound before GETE calls can be executed.
3. If INTENT=BIND, the module must be bound before GETN calls can be executed.
4. If INTENT=ACCESS, only one module can be included in a workmod. If INTENT=BIND, DDNAME and optionally MEMBER must be specified.
5. If INTENT=ACCESS, many of the options can be ignored.

Chapter 2. IEWBUFF - Binder API buffers interface assembler macro for generating and mapping data areas

This topic is specifically for users of the IEWBIND and IEWBUFF macros. It includes a complete description of the use of IEWBUFF, as well as supplemental material on the IEWBIND macro. If you are calling the binder API from languages other than assembler or IBM's internal PL/X, you must follow the instructions presented in “Invoking the binder without the macros” on page 20 together with the IEWBIND interface parameter details given in “IEWBIND function reference” on page 24.

The binder uses a standardized buffer structure to pass data to and from calling programs. Each buffer begins with a header containing the following information:

1. A doubleword containing an identifier
2. A fullword containing the length of the buffer (including the header)
3. A one-byte version identifier followed by three bytes of zeroes
4. A fullword containing the length of each entry in the buffer ('1' if the buffer holds TEXT data)
5. A fullword containing the number of bytes (for TEXT data) or records (for other data types) the buffer can hold.

This section describes how to use the IEWBUFF macro to generate and initialize buffers and to obtain and release the necessary storage. The record layouts for each buffer type are illustrated in Appendix D, “Binder API buffer formats,” on page 239.

Using the IEWBUFF macro

The IEWBUFF macro generates, initializes, and maps the data buffers used during binder processing. The IEWBUFF macro provides four functions:

- MAPBUF to generate a buffer declaration for one data type
- GETBUF to acquire storage for one buffer
- INITBUF to initialize a buffer header for one buffer
- FREEBUF to release storage for one buffer.

```
IEWBUFF TYPE={CUI | ESD | LIB | RLD | IDRU | IDRL | IDRZ | IDRB |
                    SYM | TEXT | NAME | XTLST | PINIT | PMAR}

             ,FUNC=MAPBUF
                  {,SIZE=size | BYTES=bytes}
                  [,HEADREG={headreg | USERHEADREG}]
                  [,ENTRYREG={entryreg | USERENTRYREG}]
                  [,VERSION=version]
                  [,PREFIX=prefix_chars]

             ,FUNC=GETBUF
                  [,GM_RETCODE=gm_retcode]

             ,FUNC=INITBUF

             ,FUNC=FREEBUF
                  [,FM_RETCODE=fm_retcode]
```

Figure 2. IEWBUFF function summary

© Copyright IBM Corp. 1991, 2012

MAPBUF provides a mapping macro for the buffer header and for one entry record. It also defines constant values, in the calling program, associated with the specified data type. These are used by the GETBUF, INITBUF, and FREEBUF calls to determine the interface version and number of records or bytes in the buffer.

Note: If it is necessary to have more than one copy of a particular buffer in your program (for example, both Version 1 and Version 2 buffers), you must code the PREFIX parameter to vary the generated data names.

GETBUF and INITBUF are used to prepare a buffer for use, and FREEBUF is used to release the buffer after usage. GETBUF and FREEBUF are not needed if you specifically provide storage for the buffer.

A single block of storage can be used for different buffer types if it is as large as the largest buffer type used and the buffer header is initialized for the correct data type before the buffer is used. When multiple buffers are to share the same storage you should specify the buffer capacity in bytes rather than SIZE. Some buffer types (such as ESD and RLD) vary considerably from release to release. You should specify the latest buffer version to be sure that all new binder features are available. The default is VERSION=1.

This section describes the parameters for each function call. No separate return codes, other than those generated by GETMAIN and FREEMAIN, are returned by the IEWBUFF macro. The invocation environment is the same as that of the IEWBIND macro (see “Setting the invocation environment” on page 19).

FREEBUF: Free buffer storage

The syntax of the FREEBUF call is:

```
[symbol]        IEWBUFF     FUNC=FREEBUF

                            ,TYPE={CUI | ESD | LIB | RLD | IDRU | IDRL |
                                   IDRZ | IDRB |  SYM | TEXT | NAME |
                                   XTLST | MAP | PINIT | PMAR}
                            [,FM_RETCODE=fm_retcode]
                            [,PREFIX=string]
```

FUNC=FREEBUF
Requests that the buffer storage be released and the base pointers for the buffer mappings be set to zero.

TYPE={CUI | ESD | LIB | RLD | IDRU | IDRL | IDRZ | IDRB | SYM | TEXT | NAME | XTLST | MAP | PINIT | PMAR}
Specifies a record buffer for B_CUI, B_ESD, B_LIB, B_RLD, B_IDRU, B_IDRL, B_IDRZ, B_IDRB, B_SYM, and B_MAP class data items; a byte-oriented buffer for B_TEXT and PMAR data; a record buffer for name lists (NAME) (see “GETN: Get names” on page 56); a data buffer for an extent list (B_XTLST) that is returned to the caller after a program module is loaded (see “LOADW: Load workmod” on page 69); and a record buffer for part initializers (B_PARTINIT) (see binder-defined class B_PARTINIT on page 3). ADATA and all compiler-defined text classes must use TEXT for buffer type.

FM_RETCODE=*fm_retcode* **– RX-type address or register (2-12)**
Specifies the location of a fullword that is to receive the return code from the FREEMAIN request issued by IEWBUFF. If not specified, an unconditional FREEMAIN will be issued.

PREFIX=*string*
Specifies that each generated symbol in the buffer declaration is prefixed with the provided string, followed by an underscore(_). The PREFIX value is needed for differentiating the field names of two versions of the same record type. For instance, if you use Version 2 and Version 3 of the ESD mapping in the same program, you can use "V2" and "V3" respectively, for each PREFIX. The mappings are generated with the differentiating prefixes and are used when multiple MAPBUF declarations exist for the same TYPE to identify which of the declarations should be used for this request.

Note: The PREFIX value should not be enclosed by apostrophes and must not exceed 32 bytes.

GETBUF: Get buffer storage

The syntax of the GETBUF call is:

```
[symbol]        IEWBUFF     FUNC=GETBUF

                            ,TYPE={CUI | ESD | LIB | RLD | IDRU | IDRL |
                                   IDRZ | IDRB | SYM | TEXT | NAME |
                                   XTLST | MAP | PINIT | PMAR}
                            [,GM_RETCODE=gm_retcode]
                            [,PREFIX=string]
```

FUNC=GETBUF
Requests that storage for a buffer be obtained and that the sections for the buffer header and the first buffer entry be mapped. This macro initializes the HEADREG and ENTRYREG specified by IEWBUFF FUNC=MAPBUF.

TYPE={CUI | ESD | LIB | RLD | IDRU | IDRL | IDRZ | IDRB | SYM | TEXT | NAME | XTLST | MAP | PINIT | PMAR}
Specifies a record buffer for B_CUI, B_ESD, B_LIB, B_RLD, B_IDRU, B_IDRL, B_IDRZ, B_IDRB, B_SYM, and B_MAP class data items; a byte-oriented buffer for B_TEXT and PMAR data; a record buffer for name lists (NAME) (see “GETN: Get names” on page 56); a data buffer for an extent list (B_XTLST) that is returned to the caller after a program module is loaded (see “LOADW: Load workmod” on page 69); and a record buffer for part initializers (B_PARTINIT) (see binder-defined class B_PARTINIT on page 3). ADATA and all compiler-defined text classes must use TEXT for buffer type.

GM_RETCODE=*gm_retcode* **– RX-type address or register (2-12)**
Specifies the location of a fullword that receives the return code from the GETMAIN request issued by IEWBUFF.If not specified, an unconditional GETMAIN will be issued.

PREFIX=*string*
Specifies that each generated symbol in the buffer declaration is prefixed with the provided string, followed by an underscore(_). The PREFIX value is needed for differentiating the field names of two versions of the same record type. For instance, if you use Version 2 and Version 3 of the ESD mapping in the same program, you can use "V2" and "V3" respectively, for each PREFIX. The mappings are generated with the differentiating prefixes (for example, V2_ESD_TYPE and V3_ESD_TYPE), and you can then refer to any field name in either mapping without ambiguity.

Note: The PREFIX value should not be enclosed by apostrophes and must not exceed 32 bytes.

INITBUF: Initialize buffer header

The syntax of the INITBUF call is:

```
[symbol]      IEWBUFF     FUNC=INITBUF

                          ,TYPE={CUI | ESD | LIB | RLD | IDRU | IDRL |
                                 IDRZ | IDRB | SYM | TEXT | NAME |
                                 XTLST | MAP | PINIT | PMAR}
                          [,PREFIX=string]
```

FUNC=INITBUF
Requests that the buffer header be initialized.

TYPE={CUI | ESD | LIB | RLD | IDRU | IDRL | IDRZ | IDRB | SYM | TEXT | NAME | XTLST | MAP | PINIT | PMAR}
Specifies a record buffer for B_CUI, B_ESD, B_ LIB, B_RLD, B_IDRU, B_IDRL, B_IDRZ, B_IDRB, B_SYM, and B_MAP class data items; a byte-oriented buffer for B_TEXT and PMAR data; a record buffer for name lists (NAME) (see "GETN: Get names" on page 56); a data buffer for an extent list (B_XTLST) that is returned to the caller after a program module is loaded (see "LOADW: Load workmod" on page 69); and a record buffer for part initializers (B_PARTINIT) (see binder-defined class B_PARTINIT on page 3). ADATA and all compiler-defined text classes must use TEXT for buffer type.

PREFIX=*string*
Specifies that each generated symbol in the buffer declaration is prefixed with the provided string, followed by an underscore(_). The PREFIX value is needed for differentiating the field names of two versions of the same record type. For instance, if you use Version 2 and Version 3 of the ESD mapping in the same program, you can use "V2" and "V3" respectively, for each PREFIX. The mappings are generated with the differentiating prefixes and are used when multiple MAPBUF declarations exist for the same TYPE to identify which of the declarations should be used for this request.

Note: The PREFIX value should not be enclosed by apostrophes and must not exceed 32 bytes.

MAPBUF: Map buffer declaration

The syntax of the MAPBUF call is:

```
[symbol]      IEWBUFF     FUNC=MAPBUF

                          ,TYPE={CUI | ESD | LIB | RLD | IDRU | IDRL |
                                 IDRZ | IDRB | SYM | TEXT | NAME |
                                 XTLST | MAP | PINIT | PMAR}
                          ,{SIZE=nnn | BYTES=nnnn}
                          [,HEADREG={headreg | USERHEADREG}]
                          [,ENTRYREG={entryreg | USERENTRYREG}]
                          [,VERSION={1|2|3|4|5|6}]
                          [,PREFIX=string]
```

FUNC=MAPBUF
Requests a declaration of the buffer header and its entries. This function is

required if any other IEWBUFF functions are used and may be specified only once per TYPE and PREFIX combination in the assembler source module containing the macro expansion.

TYPE={CUI | ESD | LIB | RLD | IDRU | IDRL | IDRZ | IDRB | SYM | TEXT | NAME | XTLST | MAP | PINIT | PMAR}
Specifies a record buffer for B_CUI, B_ESD, B_LIB, B_RLD, B_IDRU, B_IDRL, B_IDRZ, B_IDRB, B_SYM, and B_MAP class data items; a byte-oriented buffer for B_TEXT and PMAR data; a record buffer for name lists (NAME) (see "GETN: Get names" on page 56); a data buffer for an extent list (B_XTLST) that is returned to the caller after a program module is loaded (see "LOADW: Load workmod" on page 69); and a record buffer for part initializers (B_PARTINIT) (see binder-defined class B_PARTINIT on page 3). ADATA and all compiler-defined text classes must use TEXT for buffer type.

SIZE=*nnn* | **BYTES=***nnnn*
SIZE=*nnn* Specifies the size of the buffer in bytes for text data buffers and in records for all other buffer types.

BYTES=*nnnn* Specifies the maximum number of bytes available for the buffer, including the header, regardless of the TYPE specified. This parameter is an alternative to the SIZE parameter and is used when a single block of storage is shared by several different buffers.

Code the values for SIZE and BYTES as integer values in the range 0-2**31.

Note: Values provided here should be treated as estimates of the amount of data that can be returned. The macro will add additional space for overhead, but that, too, is only an estimate. The amount of data actually returned is always reported by the entry count in the output parameter list and the entry length as defined in the buffer header for each type of buffer.

HEADREG={*headreg* | **USERHEADREG}**
Specifies a value from 1-15 indicating the register number to equate to the base register used for the buffer header dummy section (DSECT). If the default, USERHEADREG, is used, IEWBUFF does not use a specific register to base the buffer header DSECT. Instead, you define a symbol named IEWBxyz_BASE, where xyz is 3 characters representing the TYPE, as the base register for the DSECT. The PREFIX value will be used to differentiate IEWBxyz_BASE as well as other symbols in the buffer declaration. For example, using TYPE=CUI,PREFIX=V2 will result in V2_IEWBCUI_BASE.

The following codes must be substituted for xyz in the register names, depending on the TYPE= keyword value used.

```
TYPE:   CUI   IDRB  IDRL  IDRU  IDRZ  LIB  NAME  PINIT  TEXT  XTLST  PMAR
xyz:    CUI   IDB   IDL   IDU   IDZ   LIB  BNL   PTI    TXT   XTL    PMR
```

ENTRYREG={*entryreg* | **USERENTRYREG}**
Specifies a value from 1-15 indicating the register number to equate to the base register used for the buffer entry DSECT. If the default, USERENTRYREG, is used, IEWBUFF does not use a specific register to base the buffer entry DSECT. Instead, you define a symbol named xyz_BASE, where xyz represents the TYPE (see the codes under **HEADREG**), as the base register for the DSECT. The PREFIX value will be used to differentiate xyz_BASE as well as other symbols in the buffer declaration. For example, using TYPE=CUI,PREFIX=V2 will result in V2_CUI_BASE.

VERSION=*1* | *2* | *3* | *4* | *5* | *6*
Each version of the binder API supports all earlier buffer format versions, so

you do not need to modify your program to call a more recent level of the binder API. It is recommended that new programs specify the latest VERSION available when they are written because the default is VERSION=1. Using the default, or any older version, is likely to make it impossible for the binder to return all of the data in the requested record; only compatible fields will contain returned data.

New buffer versions were introduced in the following releases:

- VERSION=1 DFSMS/MVS Release 1
- VERSION=2 DFSMS/MVS Release 3
- VERSION=3 DFSMS/MVS Release 4
- VERSION=4 OS/390® DFSMS Version 2 Release 10
- VERSION=5 z/OS Version 1 Release 3
- VERSION=6 z/OS Version 1 Release 5

PREFIX=*string*

Specifies that each generated symbol in the buffer declaration is prefixed with the provided string, followed by an underscore (_). The PREFIX value is needed for differentiating the field names of two versions of the same record type. For instance, if you use Version 2 and Version 3 of the ESD mapping in the same program, you can use "V2" and "V3" respectively, for each PREFIX. The mappings are generated with the differentiating prefixes (for example, V2_ESD_TYPE and V3_ESD_TYPE), and you can then refer to any field name in either mapping without ambiguity.

Note: The PREFIX value should not be enclosed by apostrophes and must not exceed 32 bytes.

Chapter 3. IEWBIND - Binder regular API functions

This topic describes how to use IEWBIND, the binder regular API functions. It contains a complete description of the use of IEWBIND, and each function that you can specify on the IEWBIND macro. The following topics are included:

- “Invoking the binder API”
- “Coding the IEWBIND macro” on page 23
- “IEWBIND function reference” on page 24

Invoking the binder API

This topic describes the following tasks:

- “Setting the invocation environment”
- “Loading the binder”
- “Invoking the binder using the macros” on page 20
- “Invoking the binder without the macros” on page 20
- “Binder API common return and reason codes” on page 21

Setting the invocation environment

All binder API calls associated with a given binder dialog must be issued from the same TCB.

Your program's environment must have the following characteristics before invoking the API:

- Enabled for I/O and external interrupts
- Holds no locks
- In task control block (TCB) mode
- With PSW key equal to the job step TCB key
- In primary address space mode
- In 31-bit addressing mode
- In either supervisor or problem program state
- With no FRRs on the current FRR stack.

You can call the binder in both problem program and supervisor state and in any PSW storage key.

If your program is written in a high-level programming language, you must ensure that your program is in 31-bit addressing mode before calling the IEWBIND service. If your program is written in assembler language, the IEWBIND macro takes care of any addressing mode changes in a transparent manner.

All requests are synchronous. The binder returns control to your program after the completion of the requested service. Services cannot be requested in cross-memory mode.

Loading the binder

The IEWBIND macro issues the LOAD macro for the IEWBIND entry point on the STARTD call and the DELETE macro at the completion of an ENDD call. In order to retain addressability to the binder on subsequent calls, the entry point address is saved as the first word in the dialog token. This token must not be modified in any way between binder calls.

© Copyright IBM Corp. 1991, 2012

If your program does not use the IEWBIND macro, you must cause the binder to be loaded for entry point IEWBIND. Subsequently, each function call is executed by calling the IEWBIND entry point with a parameter list.

Invoking the binder using the macros

Assembler language programs are expected to use the IEWBIND and IEWBUFF macros. Sections “Coding the IEWBIND macro” on page 23 and “IEWBIND function reference” on page 24 contain the details for coding the IEWBIND macro. Section Chapter 2, “IEWBUFF - Binder API buffers interface assembler macro for generating and mapping data areas,” on page 13 describes the coding details of the IEWBUFF macro.

The IEWBIND macro generates standard linkage code to transfer control to the binder, so register 13 must contain the address of an 18-word register save area.

Invoking the binder without the macros

If you want to invoke the binder without using the macros (for example, from nonassembler language programs), provide the services and generate the code equivalent to that provided by the macros.

Providing buffer areas

For programs that create or access module data through binder function calls (GETC, GETD, GETE, GETN, PUTD) you must acquire the buffers and create the record map for each record type you plan to access. Functions GETC, GETD, GETE, GETN and PUTD have a data buffer parameter defining the location of the buffer to be used. This location is the address of the buffer header as described in Chapter 2, “IEWBUFF - Binder API buffers interface assembler macro for generating and mapping data areas,” on page 13. See the following for a discussion of the IEWBUFF and its associated macros for the specifications:

- Chapter 2, “IEWBUFF - Binder API buffers interface assembler macro for generating and mapping data areas,” on page 13
- Appendix D, “Binder API buffer formats,” on page 239
- “Using the IEWBUFF macro” on page 13

Providing the environment

Your program must have the execution environment defined in “Setting the invocation environment” on page 19. If the addressing mode is not 31-bit, your program must switch to 31-bit addressing mode before calling the binder. Upon return from the binder, your program can restore the 24-bit addressing mode.

Obtaining the address of the binder program

You load the binder using its IEWBIND entry point and save the entry point address. If your compiler does not support a LOAD function, call an assembler subroutine to issue the LOAD macro and return the entry point address. You can use the first word of the dialog token and/or the workmod token to save the entry point address. When your program is complete, you cause the DELETE macro to be issued to delete the binder from your execution environment.

Providing the parameter lists

Your program defines and sets each variable, then creates the parameter list for each function call. The parameter list for each function call is provided in a figure at the end of the function call description. See “IEWBIND function reference” on page 24.

Don't forget to set the high-order bit in the last address of the list to the number 1. This bit signifies to the binder the end of the list of addresses.

Your program is responsible for all of the variables. You must ensure the integrity of the dialog and workmod tokens between calls. Check that all varying character strings have a halfword prefix containing the correct length of the data (not including the prefix). The 2 byte length field is a binary number which immediately precedes the data and is the length of the data only (that is, the length does not include the two bytes where it resides). See Figure 3 for a visual representation of a variable length string parameter.

Figure 3. Example of a variable length string parameter

Invoking the binder

Your program calls the binder using MVS linkage conventions:

- Register 1 contains the address of the parameter list.
- Register 13 contains the address of an 18-word register save area.
- Register 14 contains the return address.
- Register 15 contains the IEWBIND entry point address.

Upon return from the binder, you can examine the return and reason codes. Fullword return and reason codes are returned in registers 15 and 0, respectively. They are also saved in the storage areas identified by the RETCODE and RSNCODE addresses in the parameter list if those addresses are nonzero.

Binder API common return and reason codes

A return code, indicating the completion status of the requested function, is returned in register 15 and in the fullword designated by the keyword RETCODE, if provided. Return codes are interpreted as follows:

0 Successful completion of the operation.

4 Successful completion, but an unusual condition existed. One or more warning level messages have been issued, and the reason code has been set to indicate the nature of the problem or situation.

8 Error condition detected. Invalid data might have been discarded. Corrective action has been taken by the binder, but results might require inspection.

12 Severe error encountered. Requested operation could not be completed, but the dialog is allowed to continue.

16 Terminating error. Dialog could not be continued. Either a program error occurred in the binder, or binder control blocks have been damaged. The integrity of binder data cannot be assured.

A reason code is returned in register 0 and in the fullword designated by the keyword RSNCODE, if provided.

The reason code identifies the nature of the problem. It is zero if the return code was zero, a valid reason code otherwise. Reason codes are in the format 83*eegggg*. *ee* is 00 except for logic errors and abends when it is *EE*or *FF* respectively. *gggg* contains an information code.

Reason codes are described with the individual function descriptions. A few reason codes are common to many functions and described in Table 3. All the binder API reason codes, and the API function that calls them are shown in “Binder API reason codes in numeric sequence” on page 99.

See *z/OS MVS Program Management: User's Guide and Reference* for a complete list of binder return codes.

Table 3. Common binder API reason codes

Return Code	Reason Code (hex)	Explanation
00	00000000	Normal completion.
08	83000813	Data incompatible with buffer version. Request rejected.
12	83000001	Invalid workmod token. Request rejected.
12	83000002	Invalid dialog token. Request rejected.
12	83000003	Binder invoked from within user exit. Request rejected.
12	83000004	Invalid function code specified. Request rejected.
12	83000005	Invalid parameter. Request rejected.
12	83000006	A function not allowed during PUTD input mode operation was requested. The request is rejected.
12	83000007	Passed symbol contains characters outside the range acceptable to the binder.
12	83000008	Wrong number of arguments specified. Request rejected.
12	83000009	Parameter list contains invalid address, or refers to storage that is not accessible by the binder. Request rejected.
12	83000010	Parameter list is not addressable by the binder. Request rejected.
12	83000101	Improper combination of parameters. Request rejected.
16	83000050	Storage limit established by workspace option exceeded. Dialog terminated.
16	83000051	Insufficient storage available. Dialog terminated. Increase REGION parameter on EXEC statement and rerun job.
16	83000060	Operating system not at correct DFSMS/MVS level. No dialog established, function not processed.
16	83000FFF	IEWBIND module could not be loaded. Issued only by IEWBIND macro.
16	83EE2900	Binder logic error. Dialog terminated. Normally accompanied by a 0F4 abend.
16	83FF*aaa*0	Binder abend occurred. Dialog terminated. *aaa* is the system abend completion code. If the binder is invoked through one of the batch entry points, no attempt is made to intercept x37 abends (indicating out-of-space conditions in SYSLMOD).

Coding the IEWBIND macro

Each IEWBIND call contains a function code keyword, an optional version number, two optional keywords indicating where the return and reason codes are to be placed after completion of the function, and either a dialog or a workmod token. The function code identifies the requested service. Other parameters vary according to the function requested.

The VERSION parameter on each call indicates the format of the parameter list. For new programs, specify VERSION=7 to ensure that new binder features are available. If you do not specify VERSION, it defaults to VERSION=1.

IEWBIND supports inline parameter list generation (MF=S) and the list (MF=L) and execute (MF=E) forms of assembler macros. The generated parameter list is documented for each IEWBIND function call.

Defining varying character strings

Many character string variables are coded as varying-length character strings. Varying-length strings are defined for user-defined names (section, class, ddname, member, symbol, part), lists of codes that must appear as a single parameter, and other parameter values that might vary in length. Varying-length character strings have a halfword length in the first two bytes of the string that contains the current length of the string not counting the length field itself. In the function call specifications in “IEWBIND function reference” on page 24, the length value associated with the string (for example, "1024-byte varying ...") indicates the maximum data length allowed in the length field.

Defining section names

Section names must be 1 to 32767 bytes in length, consisting entirely of EBCDIC characters in the range X'41' through X'FE', plus X'0E' and X'0F' (for double-byte character set (DBCS) support.

Defining parameter lists

A parameter list is generated for each IEWBIND function call. The first three parameter addresses point to a fullword containing a halfword function code and the halfword parameter list format, an optional fullword return code, and an optional 32-bit reason code. The remaining parameters vary in number and sequence according to the requested function. Bit zero is set in the last parameter address to indicate the end of the parameter list.

Setting null values

The IEWBIND macro generates the necessary null and default values for any omitted parameters. Parameters not coded on the macro are passed as null values, rather than omitting the parameter altogether. Null values for the parameters are as follows:

Parameter	Null value
Varying length character strings	Two bytes of zeros.
Fixed length character strings	Blank
Fullword or halfword integers	Zero, unless zero is a valid value for that parameter (such as OFFSET); then use –1
Doubleword integers (RELOC)	Two words of zeros.

Parameter	Null value
Return and reason codes	Zero parameter address
Data buffers (AREA, XTLST)	Doubleword of zero
Lists (FILES, EXITS, OPTIONS)	Fullword of zero
Addresses	Zero
Tokens	Fullword or doubleword of zero depending upon the token length. DIALOG, WORKMOD, and EPTOKEN are 8 bytes; use a doubleword
WKSPACE (workspace) parameter	Doubleword of zero

IEWBIND function reference

This topic describes each function that you can specify on the IEWBIND macro and the associated parameters.

```
|                              IEWBIND       [,VERSION=1 | 2 | 3 | 4 | 5 | 6 | 7 ]
                                             [,RETCODE=retcode]
                                             [,RSNCODE=rsncode]
                                             [,MF=S]                          Macro standard form
                                             [,MF=(L,mfctrl[,{0F | 0D}])]     Macro list form
                                             [,MF=(E,mfctrl)]                 Macro execute form

                                               FUNC=ADDA                      Add alias
                                                  ,WORKMOD=workmod
                                                  ,ANAME=aname
                                                 [,ATYPE={A | S | P}]
                                                 [,ENAME=ename]
                                                 [,AMODE=amode]

                                               FUNC=ALIGNT                    Align text
                                                  ,WORKMOD=workmod
                                                  ,SECTION=section

                                               FUNC=ALTERW                    Alter workmod
                                                  ,WORKMOD=workmod
                                                  ,OLDNAME=oldname
                                                  ,ATYPE={CHANGE | C}
                                                      ,NEWNAME=newname

                                                  ,ATYPE={DELETE | D}

                                                  ,ATYPE={EXPAND | E}
                                                      ,COUNT=count
                                                      [,CLASS=class]

                                                  ,ATYPE={REPLACE | R}
                                                      ,NEWNAME=newname

                                                [,MODE={NEXT | N |IMMED | I}]

                                               FUNC=AUTOC                     Autocall
                                                  ,WORKMOD=workmod
                                                  ,{CALLIB=callib | PATHNAME=pathname}

                                               FUNC=BINDW                     Bind workmod
                                                  ,WORKMOD=workmod
                                                [,CALLIB=callib]

                                               FUNC=CREATEW                   Create workmod
                                                  ,DIALOG=dialog
                                                  ,WORKMOD=workmod
                                                  ,INTENT={BIND | B | ACCESS | A}

                                               FUNC=DELETEW                   Delete workmod
                                                  ,WORKMOD=workmod
                                                [,PROTECT={YES | Y | NO | N}]

                                               FUNC=DLLR                      DLL rename
                                                  ,WORKMOD=workmod
                                                  ,RENAMEL=renamelist
```

Figure 4. IEWBIND function call summary (Part 1 of 4)

```
FUNC=ENDD                        End dialog
   ,DIALOG=dialog
  [,PROTECT={YES | Y | NO | N}]

FUNC=GETC                        Get compile unit list
   ,WORKMOD=workmod
   [,SECTION=section]
   ,AREA=area
   ,COMPILEUNITLIST=compileunitlist
   ,CURSOR=cursor
   ,COUNT=count
FUNC=GETD                        Get data
   ,WORKMOD=workmod
   ,CLASS=class
  [,SECTION=section]
   ,AREA=area
   ,CURSOR=cursor
   ,COUNT=count
   ,RELOC=reloc
FUNC=GETE                        Get ESD
   ,WORKMOD=workmod
  [,SECTION=section]
  [,RECTYPE=rectype]
  [,{OFFSET=offset | SYMBOL=symbol}]
  [,CLASS=class]
   ,AREA=area
   ,CURSOR=cursor
   ,COUNT=count

FUNC=GETN                        Get names
   ,WORKMOD=workmod
  [,AREA=area]
   ,CURSOR=cursor
   ,COUNT=count
   ,TCOUNT=tcount
  [,NTYPE={CLASS | C | SECTION | S}]

FUNC=IMPORT                      Import symbol
   ,WORKMOD=workmod
   ,ITYPE={CODE | C | DATA | D}
   ,DLLNAME=dllname
   ,INAME=iname
   [,OFFSET=offset]
```

Figure 4. IEWBIND function call summary (Part 2 of 4)

```
FUNC=INCLUDE                        Include
   ,WORKMOD=workmod
   ,INTYPE={NAME | N}
       ,DDNAME=ddname[,MEMBER=member]
       | ,PATHNAME=pathname

   ,INTYPE={POINTER | P}
       ,DCBPTR=dcbptr
       ,DEPTR=deptr

   ,INTYPE={TOKEN | T}
       ,EPTOKEN=eptoken

   ,INTYPE={DEPTR | D}
       ,{DDNAME=ddname | DEPTR=deptr}

   ,INTYPE={SMDE | S}
       ,{DDNAME=ddname | DEPTR=deptr}

  [,ATTRIB={NO | N | YES | Y}]
  [,ALIASES={NO | N | YES | Y}]
  [,IMPORTS={NO | N | YES | Y}]

FUNC=INSERTS                        Insert section
   ,WORKMOD=workmod
   ,SECTION=section

FUNC=LOADW                          Load workmod
   ,WORKMOD=workmod
   ,EPLOC=eploc
   ,IDENTIFY={NO | N}
       ,XTLST=xtlst

   ,IDENTIFY={YES | Y}
     [,XTLST=xtlst]
     [,LNAME=lname]

FUNC=ORDERS                         Order section
   ,WORKMOD=workmod
   ,SECTION=section

FUNC=PUTD                           Put workmod data
   ,WORKMOD=workmod
   ,CLASS=class
   ,SECTION=section
  [,AREA=buffer,COUNT=count]
  [,CURSOR=cursor]
  [,NEWSECT={NO | YES}]
  [,ENDDATA={NO | YES}]
```

Figure 4. IEWBIND function call summary (Part 3 of 4)

```
FUNC=RENAME                        Rename symbol
   ,OLDNAME=oldname
   ,NEWNAME=newname

FUNC=RESETW                        Reset workmod
   ,WORKMOD=workmod
   ,INTENT={BIND | B | ACCESS |A}
  [,PROTECT={YES | Y | NO | N}]

FUNC=SAVEW                         Save workmod
   ,WORKMOD=workmod
  [,MODLIB={ddname | pathname}]
  [,SNAME=sname]
  [,REPLACE={NO | N | YES | Y}]

FUNC=SETL                          Set library
   ,WORKMOD=workmod
  [,SYMBOL=symbol]
  [,LIBOPT={CALL | C | NOCALL | N | EXCLUDE |E}]

  [,CALLIB=ddname | PATHNAME=pathname]

FUNC=SETO                          Set options
  {,DIALOG=dialog | ,WORKMOD=workmod}
  ,OPTION=option
  ,OPTVAL=optval
  [,PARMS=parms]

FUNC=STARTD                        Start dialog
   ,DIALOG=dialog
  [,FILES=files]
  [,EXITS=exits]
  [,OPTIONS=options]
  [,PARMS=parms]
  [,ENVARS=envars]

FUNC=STARTS                        Start segment
   ,WORKMOD=workmod
   ,ORIGIN=origin
  [,REGION={NO | N | YES | Y}]
```

Figure 4. IEWBIND function call summary (Part 4 of 4)

Note: List and Execute form parameters are defined in detail in the IEWBIND2 macro.

ADDA: Add alias

ADDA allows you to specify an alias to be added to a list of alias names.

The syntax of the ADDA call is:

```
[symbol]    IEWBIND    FUNC=ADDA

                       [VERSION=version]
                       [,RETCODE=retcode]
                       [,RSNCODE=rsncode]
                       ,WORKMOD=workmod
                       ,ANAME={alias | pathname}
                       [,ATYPE={A | S | P}]
                       [,ENAME=ename]
                       [,AMODE=amode]
```

FUNC=ADDA
Specifies that you are adding an alias.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

Note: If VERSION is defaulted or if specified as 1, 2, or 3, ATYPE cannot be specified as a macro keyword.

RETCODE=*retcode* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **– RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that is to receive the reason code returned by the binder.

WORKMOD=*workmod* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the workmod token returned on the CREATEW request.

ANAME={*alias* | *pathname***} – RX-type address or register (2-12)**
Specifies the location of a 1024–byte varying character string that contains the alias or alternate entry point name to be added to the directory of the output program library. This is an alias name if ATYPE=A, or a path name if ATYPE=S or ATYPE=P.

ENAME=*ename* **– RX-type address or register (2-12)**
Specifies the location of a varying character string up to 32767 bytes long that contains the name of the entry point that is to receive control when the program module is accessed by the name specified in ANAME. If the name specified for ENAME is not a defined label in the ESD, the program will be entered at its primary entry point. If no value is provided for ENAME, the value defaults to the value specified in ANAME. ENAME is ignored if ATYPE is S or P.

ATYPE={A | S | P}
Specifies what type of value is pointed to by ANAME. ATYPE is not supported if VERSION is defaulted or is specified as less then 4. ATYPE can be:
A Regular alias (default)
S The pathname for a symbolic link
P Sympath - ANAME is the pathname to be stored in all symbolic links.

AMODE=*amode* **– RX-type address or register (2-12)**
Specifies the location of a varying character string that contains the addressing mode for the entry point specified. This value overrides any other AMODE value already specified for this entry point without affecting other entry points. The values that can be specified for AMODE are 24, 31, 64, ANY, and MIN. AMODE is ignored if ATYPE is S or P. For a detailed description of addressing modes, see *z/OS MVS Program Management: User's Guide and Reference*.

Processing notes

The ADDA function has no immediate effect on the output module. Instead, the symbol is added to a list of symbols that is used to generate aliases when the module is saved. If the specified symbol appeared on an earlier ADDA call or an ALIAS control statement, a warning message is issued and the latter specification replaces the original one. This list is used when the program module is saved in a program library to update the directory of the target library.

If the name you specify matches a symbol already defined in the ESD of the program module, it is processed as an alternate entry point instead of a true alias.

Alternate entry points are not supported for program objects that reside in z/OS UNIX System Services files. If a z/OS UNIX System Services path name is specified, that name becomes a true alias of the primary entry point.

Call sequence is significant for the ADDAlias function call. If multiple ADDA calls specify the same alias or alternate entry point name, the most recent specification prevails. If a subsequent INCLUDE function call specifying ALIASES=YES includes an alias to the same name, the included specification takes precedence. To be sure that the ADDA specification is used, it should follow all INCLUDE function calls specifying ALIASES=YES. If the module contains multiple text classes, the primary and alternate entry points must be defined in the same class.

Return and reason codes

The common binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion. Symbol added to the list of aliases.
04	8E000114	An attempt to change module attributes failed. Symbol added to the list of aliases.
04	83000711	Alias name has already been assigned. This request will replace the previous request for this alias name.

Parameter list

If your program does not use the IEWBIND macro, place the address of the ADDA parameter list in general purpose register 1.

Table 4. ADDA parameter list

```
PARMLIST    DS      0F
            DC      A(ADDA)             Function code + version
            DC      A(RETCODE)          Return code
            DC      A(RSNCODE)          Reason code
            DC      A(WORKMOD)          Workmod token
            DC      A(ANAME)            Alias Name
            DC      A(ENAME)            Entry point name
            DC      A(AMODE)            amode
            DC      A(ATYPE)            (optional) alias type
ADDA        DC      H'30'               ADDA function code value
            DC      H'version'          Parameter list version number
            DC      C'A'                (optional) type of alias:
                                        'A' = regular alias:
                                        'S' = path name for a symbolic link:
                                        'P' = path name to store in the link
```

Note: X'80000000' must be added to either the AMODE parameter (for Version less than 54) or the ATYPE parameter.

ALIGNT: Align text

ALIGNT allows you to specify 4 KB page alignment (in the loaded module) for a section. If this is a section with multiple text classes, all elements in the section

(with the exception of those elements in merge classes) will be aligned on a page boundary. The alignment specification will not be preserved if the module is rebound.

The syntax of the ALIGNT call is:

```
[symbol]    IEWBIND    FUNC=ALIGNT

                       [,VERSION=version]
                       [,RETCODE=retcode]
                       [,RSNCODE=rsncode]
                       ,WORKMOD=workmod
                       ,SECTION=section
```

FUNC=ALIGNT
Specifies that the control section is aligned at a page boundary.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

RETCODE=*retcode* **— RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **— RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that is to receive the reason code returned by the binder.

WORKMOD=*workmod* **— RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the workmod token returned on the CREATEW request.

SECTION=*section* **— RX-type address or register (2-12)**
Specifies the location of a varying character string that contains the name of the control section or common section to be page aligned.

Processing notes

An ALIGNT request is valid only when the processing intent is BIND.

If the section does not appear, a message will be issued.

The *page_aligned* attribute is set for the target workmod and, when the module is saved, in the output directory.

Page alignment requests remain in effect only for the current bind. The alignment specification in the ESD record is not changed.

Return and reason codes

The common binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion. Section will be aligned during bind operation.
12	83000104	Function not allowed for INTENT=ACCESS. Request rejected.

Return Code	Reason Code	Explanation
04	83000710	Duplicate alignment request. A request to page align this section has already been processed. This request is ignored.

Parameter list

If your program does not use the IEWBIND macro, place the address of the ALIGNT parameter list in general purpose register 1.

Table 5. ALIGNT parameter list

```
PARMLIST    DS      0F
            DC      A(ALIGNT)               Function code
            DC      A(RETCODE)              Return code
            DC      A(RSNCODE)              Reason code
            DC      A(WORKMOD)              Workmod token
            DC      A(SECTION+X'80000000')  Section name and end-of-list indicator
ALIGNT      DC      H'31'                   ALIGNT function code
            DC      H'version'              Interface version number
```

ALTERW: Alter workmod

ALTERW allows you to change or delete symbols, control sections, or common areas and lengthen sections in a program module. The value specified on the MODE parameter determines whether the request is performed on all items currently in the workmod, or delayed to be performed only on the next program module included in the workmod.

The ALTERW request with MODE=NEXT should be followed by an include request for an object module or program module. If it is not, any pending alterations for the next included data set are ignored.

The syntax of the ALTERW call is:

```
[symbol]    IEWBIND    FUNC=ALTERW

                       [,VERSION=version]
                       [,RETCODE=retcode]
                       [,RSNCODE=rsncode]
                       ,WORKMOD=workmod
                       ,ATYPE={CHANGE | DELETE | EXPAND |
                       REPLACE}
                       [,MODE={IMMED | NEXT}]
                       ,OLDNAME=oldname
                       [,NEWNAME=newname]
                       [,COUNT=count]
                       [,CLASS=class]
```

FUNC=ALTERW
Specifies that you are changing or deleting a symbol or control section or lengthening a section within the workmod.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

Note: If VERSION=1 is specified for the ALTERW call, CLASS cannot be specified as a macro keyword. The parameter list ends with the COUNT parameter (with the high-order bit set). This exception is for Version 1 only.

RETCODE=*retcode* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **– RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that is to receive the reason code returned by the binder.

WORKMOD=*workmod* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the workmod token returned on the CREATEW request.

ATYPE={CHANGE | DELETE | EXPAND | REPLACE}
Specifies the type of alteration that is made on the designated workmod. The value for ATYPE can be abbreviated as: **C**, **D**, **E**, or **R**.

The alteration is performed either on the next module included in the workmod or on all modules currently in the workmod, depending on the argument specified on the MODE parameter. The possible arguments are as follows:

CHANGE
Changes an external symbol of any ESD type from OLDNAME to NEWNAME. The symbol is changed in the target module(s). Occurrences of the symbol in other modules, other workmods, or in directory entries are not affected.

If NEWNAME is already a defined symbol in the workmod, the existing NEWNAME is deleted when this function begins processing. A warning message is issued. Note that the results of the CHANGE operation can differ from those of the linkage editor in this situation,

DELETE
Deletes an external symbol in the target module(s). If you specify an entry name, the symbol definition is removed from the ESD and symbol references are unaffected. If you specify a control section or common section name, any items in the workmod with that section name are deleted. any external references in the workmod to the deleted symbol are unresolved.

EXPAND
Expands the length of the text of a section. The COUNT parameter is required and indicates the amount by which the section should be expanded. The same rules that apply to usage of the EXPAND control statement are in effect; see the EXPAND statement in *z/OS MVS Program Management: User's Guide and Reference* for more information.

REPLACE
Allows you to delete a symbol (OLDNAME) and change any references to that symbol to a new name (NEWNAME). If you specify a symbol that refers to a section, all items having that name are deleted from the workmod. If you specify a symbol, this operation is the same as the CHANGE alteration. Any external references to the deleted section within the workmod are changed to the new name.

MODE={IMMED | NEXT}
Specifies when the operation is to take place. The values are as follows:

IMMED
The operation is to take place on all modules currently in the workmod.

NEXT
The operation is to take place on the next module included in the workmod. This value is the default.

The value for MODE can be abbreviated as **I** or **N**.

OLDNAME=*oldname* **– RX-type address or register (2-12)**
Specifies the location of a varying character string that contains the section name or symbol to be changed, deleted, or replaced, or the name of the section to be expanded. If OLDNAME is left blank, it is assumed to be a reference to a blank common section. The maximum length of OLDNAME is 32767 bytes.

NEWNAME=*newname* **– RX-type address or register (2-12)**
Specifies the location of a varying character string that contains the new symbol name for a change or replace operation. NEWNAME must not contain all blanks. The maximum length of NEWNAME is 32767 bytes.

COUNT=*count* **– RX-type address or register (2-12)**
Specifies the location of a fullword that contains the number of bytes by which to lengthen a section for an expand operation.

CLASS=*class* **– RX-type address or register (2-12)**
Specifies the name of a 16-byte varying character string variable containing the class name of the item to be EXPANDed. If you specified anything other than EXPAND for the ATYPE parameter, this parameter is ignored. If CLASS is not specified, the default is B_TEXT.

Processing notes

An ALTERW request is valid only when the processing intent is BIND.

NEWNAME is required for change and replace alterations. It is ignored on delete and expand alterations.

The scope of the requested operation is the designated workmod or the next module to be included into the workmod by the binder, regardless of its source. An included module refers to the next object or program module to enter the designated workmod; the first END record (object module) or end-of-module indication (program module) delimits the scope of ALTERW. Alter Workmod does not affect any module(s) residing in other workmods or that enter the workmod following the target module.

If an ALTERW request is not followed by an INCLUDE, such that a BINDW, LOADW or SAVEW request is received and alterations are pending, the alterations are not applied to the first autocalled module but are ignored. Similarly, if one or more alterations are pending as a result of a CHANGE or REPLACE control statement encountered in an autocalled member and an end-of-file is encountered, those alterations are not applied to the next autocalled member.

ALTERW has no effect on symbols or section names appearing in previous or subsequent deferred function requests, such as ADDA, ALIGNT, INSERTS, ORDERS, or SETL.

Return and reason codes

The common binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion. Module altered or deferred request accepted.
04	83000702	OLDNAME not found. For an immediate-mode change or replace request, no ESD entries in the module contained the specified name.
04	83000706	Duplicate name. For an immediate mode request, the replacement name already exists as an external symbol in the target workmod. The old name or section will be deleted if necessary, and the requested change will be made.
08	83000550	A section for which an expand request was made is not in the target workmod. Workmod is unchanged.
08	83000551	The name on an expand request matched a symbol in workmod that was not a section name. Workmod is unchanged.
08	83000552	The name on a change or replace request is blank. Workmod is unchanged.
08	83000553	Expand request for more than 1 gigabytes was made. Workmod is unchanged.
08	83000554	The class name specified or defaulted does not exist in the section you specified. The element cannot be expanded. Workmod is unchanged.
08	83000555	Designated class is not a text class. The element cannot be expanded. Workmod is unchanged.
12	83000104	INTENT=ACCESS specified for workmod. Module could not be altered.

Parameter list

If your program does not use the IEWBIND macro, place the address of the ALTERW parameter list in general purpose register 1.

Table 6. ALTERW parameter list

```
PARMLIST    DS      0F
            DC      A(ALTERW)           Function code
            DC      A(RETCODE)          Return code
            DC      A(RSNCODE)          Reason code
            DC      A(WORKMOD)          Workmod token
            DC      A(ATYPE)            Alter type
            DC      A(MODE)             Alter mode
            DC      A(OLDNAME)          Old name
            DC      A(NEWNAME)          New name
            DC      A(COUNT)            Number of bytes
            DC      A(CLASS)            Class (optional)
ALTERW      DC      H'50'               ALTERW function code
            DC      H'version'          Interface version number
```

Table 6. ALTERW parameter list (continued)

```
ATYPE    DC    C'C'          Alter type:

                               'C' = Change
                               'D' = Delete
                               'E' = Expand
                               'R' = Replace
MODE     DC    C'N'          Alter mode:

                               'I' = Immediate
                               'N' = Next
```

Note: X'80000000' must be added to either the COUNT parameter (for Version 1) or the CLASS parameter (for Version 2 or higher).

AUTOC: Perform incremental autocall

Perform immediate (incremental) autocall, using the given library name as the CALLIB. Incremental autocall attempts to resolve any unresolved symbols at the time the call is made, using a single library or library concatenation. Incremental autocall does not cause immediate binding.

The syntax of the AUTOC call is:

```
[symbol]    IEWBIND    FUNC=AUTOC

                       [,VERSION=version]
                       [,RETCODE=retcode]
                       [,RSNCODE=rsncode]
                       ,WORKMOD=workmod
                       {,CALLIB=callib | ,PATHNAME=pathname}
```

FUNC=AUTOC
Requests incremental autocall.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

RETCODE=*retcode* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **– RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that is to receive the reason code returned by the binder.

WORKMOD=*workmod* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the workmod token returned by the binder on the CREATEW request. You must not modify this token.

CALLIB=*callib* **– RX-type address or register (2-12)**
Specifies the name of an 8-byte varying character string containing the ddname of the library or library concatenation to be searched during autocall processing. Either CALLIB or PATHNAME must be specified, but not both.

PATHNAME=*pathname* **– RX-type address or register (2-12)**
Specifies the name of a 1023-byte varying character string that contains the

path name of the z/OS UNIX System Services file to be searched during autocall processing. The path name must begin with "/" (absolute path) or "./" (relative path), followed by the path name up to a maximum of 1023 significant characters. Either PATHNAME or CALLIB must be specified, but not both.

Processing notes

If *pathname* represents a z/OS UNIX System Services file, the binder will assume that the file is a z/OS UNIX System Services archive file. If it is a z/OS UNIX System Services directory file, the file names in the directory will be used for symbol resolution during autocall.

Incremental autocall does not perform all of the normal autocall functions. Messages relating to unresolved references are not issued. RENAME control statements are not processed, and C library renames and the renames associated with the UPCASE option are not performed. The interface validation is not called.

Incremental autocall is not performed if the NCAL processing option is in effect

Return and reason codes

The binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion. Autocall processing was successful and the symbol(s) was resolved.

Parameter list

If your program does not use the IEWBIND macro, place the address of the AUTOCall parameter list in general purpose register 1.

Table 7. AUTOCall parameter list

```
PARMLIST    DS      0F
            DC      A(AUTOC)              Function code
            DC      A(RETCODE)            Return code
            DC      A(RSNCODE)            Reason code
            DC      A(WORKMOD)            Workmod token
            DC      A(CALLIB+X'80000000') Autocall library DDname or path name
AUTOC       DC      H'51'                 AUTOCall function code
            DC      H'version'            Interface version number
```

BINDW: Bind workmod

BINDW requests binding of the current workmod. Binding performs the following services:

- Resolving references between control sections
- Resolving unresolved external references from designated libraries
- Ordering sections as specified on any ORDERS and INSERTS calls
- Completing requests for page alignment
- Calculating any relocatable address constants
- Updating the RLD and ESD.

The syntax of BINDW is:

```
[symbol]    IEWBIND    FUNC=BINDW

                       [,VERSION=version]
                       [,RETCODE=retcode]
                       [,RSNCODE=rsncode]
                       ,WORKMOD=workmod
                       [,CALLIB=ddname]
```

FUNC=BINDW
Requests a bind of the workmod into a program module.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

RETCODE=*retcode* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **– RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that is to receive the reason code returned by the binder.

WORKMOD=*workmod* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the workmod token returned on the CREATEW request.

CALLIB=*ddname* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte varying character string that contains the ddname of the library or library concatenation to be used for automatic library call.

Processing notes

A BINDW request is valid only when the processing intent is BIND.

The processing rules for resolving external references are:

1. If the symbol was specified on a previous SETL call, one of the following occurs:
   - If the SETL specified a library ddname, that library should be searched. If the member could not be located, no attempt is made to resolve the symbol from the autocall libraries.
   - If the SETL specified either the *no call*or *exclude* option, the symbol remains unresolved and the ESD entry is marked accordingly.
2. If the symbol was not specified on a previous SETL call and the RES option is in effect, attempt to resolve the symbol from the link pack area.

   Note: If any symbols are resolved from the link pack area, the module cannot be saved on external storage.
3. If the above two conditions are not true, attempt to resolve the symbol from the appropriate autocall library:
   - If the CALLIB parameter was specified on the BINDW call, attempt to resolve the symbol from that library.
   - If a library was specified as a CALLIB option from STARTD or SETO, attempt to resolve the symbol from that library.
   - Otherwise, the reference is not resolved.

4. Once the module has been bound, no further modifications can be applied. To make additional changes it is necessary to save the existing module, reset the workmod, and include the saved module.

Sections are ordered in order of inclusion unless this order is overridden by INSERTS or ORDERS calls.

Return and reason codes

The binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion. Workmod has been bound.
04	83000300	Unresolved external references exist. NCAL, NOCALL or NEVERCALL specified. Workmod has been bound.
04	83000308	Unresolved external references exist. A member matching the unresolved reference was included during autocall, but did not contain an entry label of the same name. Workmod has been bound.
04	83000316	The overlay option was specified, but there is only one segment. The workmod is bound, but not in overlay format.
04	83000314	At least one valid exclusive call was found in a module bound in overlay format. The XCAL option was specified. Workmod has been bound.
04	83002497	The binder expected a symbol to be resolved from a specific library member but it was not.
08	83000301	Unresolved external references exist. The referenced symbols could not be resolved from the autocall library. Workmod has been bound.
08	83000302	Unresolved external references exist. No autocall library specified. Workmod has been bound.
08	83000303	Unresolved external references exist. The member(s) were located in the autocall library, but an error occurred while attempting to include one or more of the members. References to the member(s) that could not be included remain unresolved. Workmod has been bound.
08	83000304	The name in an insert request was not resolved, or was not resolved to a section name.
08	83000305	An ORDER request was processed for a symbol that is not a label in the ESD. Ordering of that symbol has been ignored. The workmod has been bound.
08	83000307	The module was bound successfully, but the module map and/or cross reference table could not be produced.
08	83000309	An ALIGN request was processed for a symbol that is not a label in the ESD. Alignment of that symbol has been ignored. The workmod has been bound.
08	83000310	One or more alteration requests were pending upon entry to autocall. The alterations were ignored. Workmod has been bound.
08	83000311	Workmod has more than one segment, but OVLY was not specified. The overlay structure was ignored, but the workmod has been bound.

Return Code	Reason Code	Explanation
08	83000313	A V-type address constant of less than four bytes, and that references a segment other than the resident segment, has been found in an overlay structure. Workmod has been bound.
08	83000315	At least one invalid exclusive call was found in a module bound in overlay format. Workmod has been bound, but the adcon for the invalid call will not be properly relocated.
08	83000317	At least one valid exclusive call was found in a module bound in overlay format. Workmod has been bound.
08	83000318	There are no calls or branches from the root segment of an overlay module to a segment lower in the tree structure. Other segments cannot be loaded. Workmod has been bound.
08	83000321	Overlay specified with COMPAT=PM2 or COMPAT=PM3. Overlay is ignored.
08	83000501	One or more control statements were included during autocall processing. The statements have been ignored.
08	83000816	Classes C_WSA and C_WSA64 are both present in the module. z/OS Language Environment does not support the presence of these two classes in the same program object, and the resulting module will not execute correctly.
08	83002623	Unexpected return code from the RACF call.
08	83002624	Unexpected error from SIGCLEAN.
08	83002481	The instruction address or the target address is not even.
08	83002492	The operand of the instruction exceeds the destination range.
08	83002493	A relative immediate instruction with multiple external symbols is encountered.
08	83002494	Certificate setup problem.
08	83002495	The binder could not resolve a weak reference used by a relative immediate address constant.
08	83002496	A relative immediate address constant references an unresolved symbol. This is not supported.
12	83000104	INTENT=ACCESS specified for workmod. Module could not be rebound.
12	83000312	There are no sections or only zero-length sections in the root segment of an overlay module, and the module probably cannot be executed. Workmod has been bound.
12	83000320	An autocall library is unusable. Either it could not be opened or the directory could not be processed. Autocall processing continues without using this library.
12	83000322	Conflicting attributes encountered within a class. The module cannot be bound.
12	83000415	Module contains no ESD data, and could not be bound.
12	83000719	Module contained no text after being bound, and is probably not executable. Processing continues.

Parameter list

If your program does not use the IEWBIND macro, place the address of the BINDW parameter list in general purpose register 1.

Table 8. BINDW parameter list

```
PARMLIST    DS      0F
            DC      A(BINDW)                Function code
            DC      A(RETCODE)              Return code
            DC      A(RSNCODE)              Reason code
            DC      A(WORKMOD)              Workmod token
            DC      A(DDNAME+X'80000000')   Library ddname and end-of-list indicator
BINDW       DC      H'70'                   BINDW function code
            DC      H'version'              Interface version number
```

CREATEW: Create workmod

CREATEW initializes a workmod and initializes the module options to the defaults for the dialog. CREATEW also specifies the processing intent that determines the functions that can be performed on the workmod.

The syntax of the CREATEW call is:

```
[symbol]        IEWBIND     FUNC=CREATEW

                            [,VERSION=version]
                            [,RETCODE=retcode]
                            [,RSNCODE=rsncode]
                            ,WORKMOD=workmod
                            ,DIALOG=dialog
                            ,INTENT={BIND | ACCESS}
```

FUNC=CREATEW
Specifies that a workmod is created and initialized.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

RETCODE=*retcode* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **– RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that is to receive the reason code returned by the binder.

WORKMOD=*workmod* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte area that is to receive the workmod token for this request.

DIALOG=*dialog* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains a dialog token for which the workmod is requested. The dialog token is obtained using the STARTD call and must not be modified.

INTENT={BIND | ACCESS}
Specifies the range of binder services that can be requested for this workmod. The values are as follows:

BIND
Specifies that the processing intent for this workmod is bind. The workmod will be bound and all binder functions can be requested.

ACCESS
Specifies that the processing intent for this workmod is access. The workmod will not be bound, and no services that alter the size or structure of the program module can be requested. See “Processing intents” on page 10 for a list of services that are not allowable.

The value for INTENT can be abbreviated as **B** or **A**.

Processing notes
None.

Return and reason codes
The common binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion. Workmod and workmod token created.

Parameter list
If your program does not use the IEWBIND macro, place the address of the CREATEW parameter list in general purpose register 1.

Table 9. CREATEW parameter list

```
PARMLIST    DS      0F
            DC      A(CREATEW)              Function code
            DC      A(RETCODE)              Return code
            DC      A(RSNCODE)              Reason code
            DC      A(DIALOG)               Dialog token
            DC      A(WORKMOD)              Workmod token
            DC      A(INTENT+X'80000000')   Processing intent and end-of-list indicator
CREATEW     DC      H'10'                   CREATEW function code
            DC      H'version'              Interface version number
INTENT      DC      CL1'A'                  Processing intent

                                              'A' = Access
                                              'B' = Bind
```

DELETEW: Delete workmod
DELETEW deletes a workmod. You must issue either the SAVEW or LOADW function call before the DELETEW unless **PROTECT=NO** has been specified. DELETEW resets the workmod token to the null state.

The syntax of the DELETEW call is:

```
[symbol]        IEWBIND     FUNC=DELETEW

                            [,VERSION=version]
                            [,RETCODE=retcode]
                            [,RSNCODE=rsncode]
                            ,WORKMOD=workmod
                            [,PROTECT={YES | NO}]
```

FUNC=DELETEW
Specifies that a workmod is deleted.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

RETCODE=*retcode* **— RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **— RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that is to receive the reason code returned by the binder.

WORKMOD=*workmod* **— RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the workmod token for this request.

PROTECT={YES | NO}
Specifying **PROTECT=N** allows the binder to delete a workmod that has been altered but not yet saved or loaded.

The value for **PROTECT** can be abbreviated as **Y** or **N**.

Processing notes

The binder is sensitive to the state of the DCB pointed to by the DCBPTR in an INCLUDE call. The DCB must not be closed and reopened while the binder accesses the corresponding data set during a dialog. Once it is opened initially for an INCLUDE call, it must remain open until after the binder's ENDD call takes place.

Note that if you do alter your DCB as described above, using DELETEW is not enough to reaccess your data set at a later time during the same binder dialog. This only causes the data set's information to remain with the dialog, and such information is no longer valid once the DCB is closed. An attempt to reuse the altered DCB in the same binder dialog might produce unpredictable results. To avoid this, end your dialog (ENDD) and start a new one (STARTD).

Return and reason codes

The common binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion. Workmod has been deleted.
08	83002624	Unexpected error from SIGCLEAN.
12	83000707	The workmod was in an altered state, and PROTECT=YES was specified or defaulted. The delete request is rejected.

Parameter list

If your program does not use the IEWBIND macro, place the address of the DELETEW parameter list in general purpose register 1.

Table 10. DELETEW parameter list

```
PARMLIST    DS      0F
            DC      A(DELETEW)          Function code
            DC      A(RETCODE)          Return code
            DC      A(RSNCODE)          Reason code
```

Table 10. DELETEW parameter list (continued)

```
            DC      A(WORKMOD)              Workmod token
            DC      A(PROTECT+X'80000000')  Protect flag and end-of-list indicator
DELETEW     DC      H'15'                   DELETEW function code
            DC      H'version'              Interface version number
PROTECT     DC      C'Y'                    Protection flag

                                              'Y' = Yes
                                              'N' = No
```

DLLR: Rename DLL modules

DLLRename allows you to rename a list of DLL names. When a module has been bound with processing option DYNAM(DLL) in effect, a table containing information about imported and exported symbols is created. The information about imported symbols includes the name of the DLL from which those symbols are to be imported. The DLLR API takes a list of existing and replacement DLL names and makes any necessary substitutions in said table.

The syntax of the DLLR call is:

```
[symbol]    IEWBIND    FUNC=DLLR

                       [,VERSION=version]
                       [,RETCODE=retcode]
                       [,RSNCODE=rsncode]
                       ,WORKMOD=workmod
                       ,RENAMEL=renamel
```

FUNC=DLLR
Requests the DLL rename function.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

RETCODE=*retcode* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **– RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that is to receive the reason code returned by the binder.

WORKMOD=*workmod* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the workmod token returned on the CREATEW request. You must not modify this token.

RENAMEL=*renamel* **– RX-type address or register (2-12)**
Specifies the name of a list of fullword addresses. The list consists of a count field, followed by one or more field trios, consisting of an old DLL name, followed by a new DLL name, followed by a code that indicates whether or not the binder successfully renamed the old name to the new name. (Figure 5 on page 45 shows the list format).

The 4-byte address of each name in the list points to a varying-length character string, which can be up to 255 bytes in length. The first two bytes in such string

indicate the length of the string, excluding the first two bytes. The 255 character length is provided for the support of z/OS UNIX System Services files, primary DLL names in PDSE libraries.

The 4-byte address of each rename code points to a full word field.

```
Off  Len
0    4     A(COUNT)           Number of trios
4    4     A(OLDNAME_1)
8    4     A(NEWNAME_1)
12   4     A(RENAME_CODE_1)
16   4     A(OLDNAME_2)
20   4     A(NEWNAME_2)
24   4     A(RENAME_CODE_2)
           ...
```

Figure 5. Rename list

Processing notes

The binder scans the DLL member names in the Import/Export table of the current workmod. If one of the DLL members in the binder table matches an OLDNAME_x entry in the passed rename list, the DLL name in the table is replaced with the corresponding new name from the list. When this happens, the RENAME_CODE_x is set to zero, meaning that the renaming function was successful for the corresponding name pair in the list. Otherwise, the rename code is set to 4 (warning), meaning that the renaming did not take place for said name pair.

This API provides the functional equivalent of the IBM® C/C++ DLLRENAME utility in support of DLL processing by the binder.

In an application, you might want to verify whether an Import/Export table exists before attempting the DLLR call. You can do so by coding a binder GETData call, with CLASS=B_IMPEXP and a TEXT buffer (refer to GETData and IEWBUFF in this chapter). GETData indicates whether the class (the Import/ Export table) exists or is empty. Since DLL support was not added until DFSMS Version 1 Release 4, program objects produced in earlier releases will not contain Import/Export tables.

When you bind a module in batch mode and specify the MAP processing option, a class entry of B_IMPEXP in the output listing reveals the existence of an Import/Export table.

Note: DLLR only affects the member names stored in the Import/Export table and does not affect the external symbols in the ESD or the directory entries for the module. DLLR does not cause the module to be rebound.

Return and reason codes

The binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion. The call completed successfully. However, you still need to verify your RENAMEList to ensure that the DLL names were indeed changed. Refer to the DLLR processing notes and the format of the DLLRename parameter list to understand what you need to verify.

Return Code	Reason Code	Explanation
04		The module that you are processing does not contain an Import/Export table; therefore, there are no DLL names to rename. Refer to the DLLR processing notes for more details.

Parameter list

If your program does not use the IEWBIND macro, place the address of the DLLRename parameter list in general purpose register 1.

Table 11. DLLRename parameter list

```
PARMLIST    DS      0F
            DC      A(DLLR)                 Function code
            DC      A(RETCODE)              Return code
            DC      A(RSNCODE)              Reason code
            DC      A(WORKMOD)              Workmod token
            DC      A(RENAMEL+X'80000000')  Rename list & end-of-list indicator
DLLR        DC      H'17'                   DLLR function code value
            DC      H'version'              Interface version number
```

ENDD: End dialog

ENDD ends the specified dialog and releases all associated storage resources. Attached workmods are deleted, data sets are closed, storage obtained on behalf of the caller is released, and the dialog token is invalidated.

The syntax of the ENDD call is:

```
[symbol]    IEWBIND    FUNC=ENDD

                       [,VERSION=version]
                       [,RETCODE=retcode]
                       [,RSNCODE=rsncode]
                       ,DIALOG=dialog
                       [,PROTECT={YES | NO}]
```

FUNC=ENDD
Specifies that a dialog is ended.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

RETCODE=*retcode* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **– RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that is to receive the reason code returned by the binder.

DIALOG=*dialog* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the token for the dialog to be terminated.

PROTECT={YES | NO}
Specifying **PROTECT=NO** allows the binder to end the dialog even if any remaining workmods have been altered but not yet saved or loaded.

The value for **PROTECT** can be abbreviated as **Y** or **N**. YES is the default.

Processing notes
None.

Return and reason codes
The common binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion. Dialog ended normally.
04	83000700	One or more workmods were in an active state but they were not protected (PROTECT=NO in ENDDialog). Dialog ended normally.
08	83000704	An unexpected condition occurred while ending the dialog. The dialog is terminated, but some resources might not have been released.
12	83000708	One or more workmods were in an "active" state, and PROTECT=YES was specified or defaulted. The dialog is not terminated.

Parameter list
If your program does not use the IEWBIND macro, place the address of the ENDD parameter list in general purpose register 1.

Table 12. ENDD parameter list

```
PARMLIST    DS      0F
            DC      A(ENDD)                 Function code
            DC      A(RETCODE)              Return code
            DC      A(RSNCODE)              Reason code
            DC      A(DIALOG)               Dialog token
            DC      A(PROTECT+X'80000000')  Protection flag and end-of-list indicator
ENDD        DC      H'5'                    ENDD function code
            DC      H'version'              Interface version number
PROTECT     DC      CL1'Y'                  Protection flag

                                              'Y' = Yes
                                              'N' = No
```

GETC: Get compile unit list
GETC returns data which is mapped to a new CUI buffer format (Version 6). The **COMPILEUNITLIST** parameter determines which data is returned.

The syntax of the GETC call is:

```
[symbol]    IEWBIND    FUNC=GETC

                       ,VERSION=version
                       [,RETCODE=retcode]
                       [,RSNCODE=rsncode]
                       ,WORKMOD=workmod
                       [,COMPILEUNITLIST=compileunitlist]]
                       ,AREA=buffer
                       ,CURSOR=cursor
                       ,COUNT=count
```

FUNC=GETC
Requests that data from items in a workmod be returned to a specified location.

VERSION=*6*
Specifies the version of the parameter list to be used (6 or higher).

RETCODE=*retcode* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **– RX-type address or register (2-12)**
Specifies the location of a 4-byte field that is to receive the reason code returned by the binder. Reason codes are documented as a sequence of 8 hexadecimal digits.

WORKMOD=*workmod* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the workmod token for this request.

COMPILEUNITLIST=*compileunitlist*
Determines which data is returned. If **COMPILEUNITLIST** is specified, one record for each compile unit in a list of compile units will be returned. If **COMPILEUNITLIST** is omitted, one record of each of all compile units will be returned. The header record, the first compile unit record, is built when the cursor is zero.

The compile unit list is a structure:

```
Count    DC    F'5'   List with 5 entries
List     DS    5F     Returned by GETN
```

Note: *compileunitlist* must be composed of values returned in BFNL_6_SECT_CU resulting from a GETN TYPE=SECTION,VERSION=6 API call.

When **INTENT=ACCESS** is specified in the CREATEW API call, information about the input module (the target module of the GETC call) is placed in the header record. This information includes the program object version and the source of the input module (data set name or path name, ddname, and member name).

AREA=*buffer* **– RX-type address or register (2-12)**
Specifies the location of a CUI buffer to receive the data. The binder returns data until either this buffer is filled or the specified items have been completely moved. See Chapter 2, “IEWBUFF - Binder API buffers interface assembler macro for generating and mapping data areas,” on page 13 for information on buffer handling.

CURSOR=*cursor* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that contains the position within the items where the binder begins processing. Specifying a zero causes the binder to return the header record, the first compile unit record. The information is provided on the DASD location of the program object. The cursor value is modified before returning to the caller.

When no compile unit list is provided, the cursor is an index into an ordered list of all CUI entries that can be returned. If the application does not modify the cursor during the retrieval process, multiple calls return all CUI records in the order by CU number because the buffer is full. When a compile unit list is provided, the cursor is an index into that application-provided list. CUI records are returned in the order specified in the CU list. If the application still does not modify the cursor during the retrieval process, multiple calls continue with

subsequent entries in the list because the buffer is full. End of data is signalled when the end of the application-provided list is reached.

COUNT=*count* **— RX-type address or register (2-12)**
Specifies the location of a fullword that receives the number of CUI records returned by the binder.

Processing notes

The CURSOR value identifies an offset into the requested data beginning with 0. It is both an input and output parameter. On input to the service, the cursor specifies the first byte to return. On exit from the service, it is updated to the next byte for continued sequential retrieval if not all data has yet been retrieved.

For load modules and program object formats at a compatability level prior to z/OS V1R5 , a compile unit is the same as a section. For z/OS V1R5 compatible modules, a compile unit corresponds to a single object module. Each compile unit in a workmod is assigned a unique number; however, this assigned number may change when a module is rebound. Furthermore, the compile unit number will be zero for all binder generated sections (IEWBLIT or section 1, for example).

Return and reason codes

The common binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion. There might be additional data that did not fit in the buffer.
04	83000800	End of data. Some data might have been returned in the buffer, but no more is available.
04	83000801	No section names exist. No data was returned.
04	83000810	Cursor is negative or beyond the end of the specified item. No data was returned.
08	83002342	Some of the passed compile unit numbers do not exist in workmod. Data for the valid compile units is returned.
12	83000102	Workmod was in an unbound state.

Parameter list

If your program does not use the IEWBIND macro, place the address of the GETC parameter list in general purpose register 1.

Table 13. GETC parameter list

```
PARMLIST    DS      0F
            DC      A(GETC)                 Function code
            DC      A(RETCODE)              Return code
            DC      A(RSNCODE)              Reason code
            DC      A(WORKMOD)              Workmod token
            DC      A(CULIST)               Compile unit list
            DC      A(BUFFER)               Data buffer
            DC      A(CURSOR)               Starting position
            DC      A(COUNT+X'80000000')    Data count
GETC        DC      H'64'                   GETC function code
            DC      H'6'                    Interface version number
```

GETD: Get data

GETD returns data from items in a workmod. The values of the **CLASS** and **SECTION** parameters determine which item is returned. If **SECTION** is omitted, all sections are returned as a single unit. This service can only be performed on a bound workmod.

The syntax of the GETD call is:

```
[symbol]    IEWBIND    FUNC=GETD

                       [,VERSION=version]
                       [,RETCODE=retcode]
                       [,RSNCODE=rsncode]
                       ,WORKMOD=workmod
                       ,CLASS=class
                       [,SECTION=section]
                       ,AREA=buffer
                       ,CURSOR=cursor
                       ,COUNT=count
                       [,RELOC=reloc]
```

FUNC=GETD
Requests that data from items in a workmod be returned to a specified location.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

RETCODE=*retcode* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **– RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that is to receive the reason code returned by the binder.

WORKMOD=*workmod* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the workmod token for this request.

CLASS=*class* **– RX-type address or register (2-12)**
Specifies the location of a 16-byte varying character string containing a class name. The class name might have been defined by the binder, a compiler, or an end user. See “Understanding binder programming concepts” on page 1 for binder class names. B_PMAR is also accepted as a class name, although it is not an actual class in a binder workmod.

SECTION=*section* **– RX-type address or register (2-12)**
Specifies the location of a varying character string that contains the name of the section to be processed. If omitted, this defaults to a concatenation of all sections in the specified class. If the processing intent is bind, the sections are ordered by virtual address. If the processing intent is access, they are returned in the same order that they were included in the workmod.

AREA=*buffer* **– RX-type address or register (2-12)**
Specifies the location of a buffer to receive the data. The binder returns data until either this buffer is filled or the specified items have been completely moved. See Chapter 2, “IEWBUFF - Binder API buffers interface assembler macro for generating and mapping data areas,” on page 13 for information on buffer handling.

CURSOR=*cursor* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that contains the position within the item(s) where the binder should begin processing. Specifying a zero for the argument causes the binder to begin processing at the start of the item. The cursor value is specified in bytes for items in the TEXT class, in records for all other classes. The value is relative to the start of the item. The cursor value is modified before returning to the caller.

COUNT=*count* **– RX-type address or register (2-12)**
Specifies the location of a fullword that is to receive the number of bytes of TEXT or the number of entries returned by the binder.

RELOC=*reloc*
Specifies a base address to be used for relocation. You can only use this parameter with **VERSION**=6 or higher. You will need to know the load segment for the data you are requesting. You can map text classes into load segments using GETN. *reloc* is a single 8–byte address. The relocation address will relocate the adcons in the returned text buffer as though the program segments had been loaded at the designated address. If you do not use the **RELOC** parameter, it should set to zero.

Processing notes

The CURSOR value identifies an index into the requested data beginning with 0 for the first data item. Each of the buffer formats defined in Appendix D, “Binder API buffer formats,” on page 239contains an entry length field in its header. Multiplying the cursor value by the entry length provides a byte offset into the data. Note that CUI, LIB, PMAR, and text data is always treated as having entry length 1. The CURSOR value is both an input and output parameter. On input to the service, the cursor specifies the first item to return. On exit from the service, it is updated to an appropriate value for continued sequential retrieval if not all data has yet been retrieved. For text data, this may or may not be the next byte after the last one returned, because pad bytes between sections and uninitialized data areas within sections may have been skipped. Any data skipped should be treated by the calling application as containing the fill character (normally X'00').

On the next GETD request, the binder begins processing where the last request left off.

If you interrupt a series of successive GETD calls, you should reset the value of the cursor before continuing. Otherwise, the cursor value might be invalid and the results of a GETD request are unpredictable.

If a section name is not passed on a GETD API invocation for a text class and the target is an overlay module, the cursor is interpreted as an offset into the module and laid out sequentially in segment order, using the alignment as specified in the object modules.

Return and reason codes

The common binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion.
04	83000800	Normal completion. Some data might have been returned in the buffer, and an end-of-data condition was encountered. There is no message associated with this condition.

Return Code	Reason Code	Explanation
04	83000801	The requested item did not exist or is empty. No data has been returned.
08	83000750	The buffer is not large enough for one record. No data is returned.
08	83000813	The buffer version is not compatible with the module content. No data is returned.
08	83002349	Not all adcons were successfully relocated. This condition could occur because relocation addresses for all the segments were not passed, or because the adcon length was insufficient to contain the address.
12	83002379	Binder encountered a bad cursor for class B_PARTINIT and processing has been stopped.
12	83000102	Workmod was in an unbound state. GETD request could not be processed.
12	83002375	The class was not a text class.

Parameter list

If your program does not use the IEWBIND macro, place the address of the GETD parameter list in general purpose register 1.

Table 14. GETD parameter list

```
PARMLIST    DS      0F
            DC      A(GETD)             Function code
            DC      A(RETCODE)          Return code
            DC      A(RSNCODE)          Reason code
            DC      A(WORKMOD)          Workmod token
            DC      A(CLASS)            Class name
            DC      A(SECTION)          Section name
            DC      A(BUFFER)           Data buffer
            DC      A(CURSOR)           Starting position
            DC      A(COUNT+X'80000000') Data count and end-of-list indicator
            DC      A(RELOC)            Relocation address
GETD        DC      H'61'               GETD function code
            DC      H'version'          Interface version number
```

GETE: Get ESD data

GETE returns data from ESD items. GETE must be used on a bound workmod. Four optional parameters allow you to specify selection criteria for the ESD items to be returned: section name, ESD record type, offset in the section or module, and symbol name. Only ESD records that meet all of the selection criteria will be returned. Multiple selection criteria can be specified to retrieve exactly the records you need.

The syntax of the GETE call is:

```
[symbol]    IEWBIND    FUNC=GETE

                       [,VERSION=version]
                       [,RETCODE=retcode]
                       [,RSNCODE=rsncode]
                       ,WORKMOD=workmod
                       [,SECTION=section]
                       [,RECTYPE=rectype]
                       [,CLASS=class]
                       [,{OFFSET=offset | SYMBOL=symbol}]
                       ,AREA=buffer
                       ,CURSOR=cursor
                       ,COUNT=count
```

FUNC=GETE
Requests that data from ESD items in a workmod be returned to a specified location.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

Note: If VERSION=1 is specified for the GETE call, CLASS cannot be specified as a macro keyword. The parameter list ends with the COUNT parameter (with the high-order bit set). This exception is for Version 1 only.

RETCODE=*retcode* **— RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **— RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that is to receive the reason code returned by the binder.

WORKMOD=*workmod* **— RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the workmod token for this request.

SECTION=*section* **— RX-type address or register (2-12)**
Specifies the location of a 16-byte varying character string that contains the name of the section to be processed. This argument can be set to blanks to indicate blank common area. Sections will be retrieved in the same order that they were included in the workmod.

The default value is all sections. If this parameter is specified, only the indicated section is searched.

RECTYPE=*rectype* **— RX-type address or register (2-12)**
Specifies the location of a varying character string that contains a list of the ESD record types to be returned. If you do not specify this argument, all record types are returned.

Record types must be identified by one- or two-character codes, separated by commas and enclosed in parentheses. Embedded blanks are not allowed. Valid record types are:

SD Section definition
ED Element definition
LD Label definition
PD Part definition

PR Part reference
ER External reference
CM Common
ST Segment table
ET Entry table
DS Dummy section definition
CM Common section definition
ET ENTAB
ST SEGTAB
PC Private code section definition
WX Weak external reference

In addition, you can use a generic code to reference more than one ESD type:
S Section definition records (SD, CM, ST, ET, PC, and DS)
U Unresolved external references (ER, ESD_STATUS=unresolved)

CLASS=*class* **– RX-type address or register (2-12)**
Specifies the location of a 16-byte varying character string containing the name of the text class referenced by the ESD record to be selected. If class has not been specified, ESD records are returned without regard to class.

OFFSET=*offset* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that contains the offset within the specified section. If a section name has not been specified, a module offset is assumed. If you specify OFFSET you cannot specify SYMBOL but must specify CLASS.

SYMBOL=*xsymbol* **– RX-type address or register (2-12)**
Specifies the location of a varying character string that contains a symbol to be used as a selection criterion. If you specify SYMBOL you cannot specify OFFSET.

If neither **OFFSET** nor **SYMBOL** is provided, processing begins at the start of the item.

AREA=*buffer* **– RX-type address or register (2-12)**
Specifies the location of a buffer to receive the data. This buffer must be allocated and initialized in ESD format. See Chapter 2, “IEWBUFF - Binder API buffers interface assembler macro for generating and mapping data areas,” on page 13 for information on buffer handling.

CURSOR=*cursor* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that indicates the position within the section or module where the binder should begin processing. Specifying a zero for this argument causes the binder to begin processing at the first ESD entry. Offsets are specified in records and are relative to the start of the selected ESD item(s).

COUNT=*count* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer in which the binder will store the number of entries it has returned.

Processing notes

The binder returns ESD records that meet the selection criteria specified on the call:

- If SECTION is specified, only that section of the ESD will be searched. All sections is the default.
- If RECTYPE is specified, only ESD records of the types appearing in the supplied list are returned.

- If OFFSET is specified and rectype="S", the ESD record for the control section (or common area) containing the specified offset, is returned for buffer version 1. The SD record mapping in other buffer versions does not contain an offset and no records will be returned. If OFFSET is specified and rectype="LD", then all LD ESD records for the symbols defined at or before that location (within the containing section) will be returned.
- If SYMBOL is specified, all ESD records of the type(s) specified with that symbol name are returned. If CLASS is specified, only ESD records that define locations in that class are returned. Some records, such as SD and ER, are not associated with any class and are never returned if class is specified.

Note: Processing of the ESD records returned by a GETE call should not make assumptions about the order of the returned ESD records. For example, such processing should not assume that LD type ESD records are returned in the order of their offsets in the section.

The CURSOR value identifies an index into the requested ESD data beginning with 0 for the first ESD. The ESD buffer formats defined in Appendix D, "Binder API buffer formats," on page 239 contain an entry length field in their headers. Multiplying the cursor value by the entry length provides a byte offset into the data. CURSOR is an input and output parameter. On input to the service, the cursor specifies the first record to return. On exit from the service, it is updated to the index of the next sequential ESD if not all data has yet been retrieved.

The binder will typically return multiple entries in a single call, depending on the size of the buffer. Data is reformatted, if necessary, to conform to the version identified in the caller's buffer. The COUNT parameter is set to the number of records actually returned in the buffer.

The ESD buffer formats defined in Appendix D, "Binder API buffer formats," on page 239 contain a record length field in their headers giving the length of each ESD record. This provides a way for the caller to index through the returned records or to access a specific record in the returned data buffer.

In some cases where OFFSET is specified and the parameter list is version 6 or less, return code 0 and reason code 0 will be returned on an end-of-data condition. The version 7 API call will always return return code 4 and reason code 83000800 on an end-of-data condition, while the COUNT may be non-zero indicating that data was returned.

Return and reason codes

The common binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion.
04	83000705	The specified symbol could not be located in the workmod. No data is returned in the buffer.
04	83000800	An end-of-data condition was detected. Some data might have been returned in buffer. There is no message associated with this condition.
04	83000801	The requested item was not found in the workmod, or was empty, or no records met the specified criteria. No data returned.

Return Code	Reason Code	Explanation
04	83000812	The specified offset was negative or beyond the end of the designated item or module. No data is returned in the buffer.
12	83000101	OFFSET and SYMBOL have both been specified. Request rejected.
12	83000102	Workmod is unbound. GETE request rejected.

Parameter list

If your program does not use the IEWBIND macro, place the address of the GETE parameter list in general purpose register 1.

Table 15. GETE parameter list

```
PARMLIST    DS      0F
            DC      A(GETE)             Function code
            DC      A(RETCODE)          Return code
            DC      A(RSNCODE)          Reason code
            DC      A(WORKMOD)          Workmod token
            DC      A(SECTION)          Section name
            DC      A(RECTYPE)          ESD record type(s)
            DC      A(OFFSET)           Offset in module or section. If not a selection
                                        criterion, set to -1.
            DC      A(SYMBOL)           Symbol name
            DC      A(BUFFER)           Data buffer
            DC      A(CURSOR)           Starting position
            DC      A(COUNT)            Data count
            DC      A(CLASS)            Text class
GETE        DC      H'62'               GETE function code
            DC      H'version'          Interface version number
RECTYPE     DC      H'7',CL7'(SD,CM)'   Sample varying string
```

Note: X'80000000' must be added to either the COUNT parameter (for Version 1) or the CLASS parameter (for Version 2 or higher).

GETN: Get names

GETN returns the names of each section or class in the workmod, a count of the total number of sections or classes, and the compile unit (CU) numbers for each section. The names returned also include names generated by the binder to represent private code sections, unnamed common, SEGTAB and ENTAB sections for overlay programs, and any other sections created by the binder. GETN can only be performed on a bound workmod.

The syntax of the GETN call is:

```
[symbol]    IEWBIND    FUNC=GETN

                       [,VERSION=version]
                       [,RETCODE=retcode]
                       [,RSNCODE=rsncode]
                       ,WORKMOD=workmod
                       [,AREA=buffer]
                       ,CURSOR=cursor
                       ,COUNT=count
                       ,TCOUNT=tcount
                       [,NTYPE={SECTION | CLASS}]
```

FUNC=GETN
Specifies that a count of the number of sections in a workmod and, optionally, the names of each section, be returned to a specified location.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

Note: This version must match the version you specify with the IEWBUFF macro when you define the buffer passed on this call.

RETCODE=*retcode* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **– RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that is to receive the reason code returned by the binder.

WORKMOD=*workmod* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the workmod token for this request.

AREA=*buffer* **– RX-type address or register (2-12)**
Specifies the location of a buffer to receive the names. This buffer must be in the format for section names (TYPE=NAME). See Chapter 2, “IEWBUFF - Binder API buffers interface assembler macro for generating and mapping data areas,” on page 13 and Appendix D, “Binder API buffer formats,” on page 239 for information on buffer definition.

Section names will be moved until either the buffer is filled or all names have been moved. This keyword is optional. If it is not specified, only the number of section names in the workmod will be returned.

CURSOR=*cursor* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that contains the position relative to the start of the list of names where the binder should begin processing. Specifying a zero for this argument causes the binder to begin processing at the beginning of the list. Offsets are specified in records and are relative to the start of the list. The cursor value is modified before returning to the caller.

COUNT=*count* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer in which the binder will indicate the number of names actually returned in the buffer.

TCOUNT=*tcount* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer in which the binder will indicate the total name count. TCOUNT indicates the total number of sections or classes in the workmod, not just those returned in the buffer.

NTYPE={SECTION | CLASS}
Specifies the type of names to be returned and counted. **SECTION** causes the names of all sections in the workmod, including special sections, to be returned. In addition, the compile unit CU numbers are provided for buffer version 6 or higher. **CLASS** causes the names of all classes in the workmod containing data to be returned. The value for NTYPE can be abbreviated as **S** or **C**. **SECTION** is the default.

Processing notes

The CURSOR value identifies an index into the requested data beginning with 0 for the first name list entry. The name list buffer formats defined in Appendix D, “Binder API buffer formats,” on page 239 contain an entry length field in their headers. Multiplying the cursor value by the entry length provides a byte offset into the data. CURSOR is both an input and output parameter. On input to the service, the cursor specifies the first item to return. On exit from the service, it is updated to the index of the next sequential name list entry if not all entries have yet been retrieved.

Return and reason codes

The common binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion.
04	83000800	An end-of-data condition was detected. Some data might have been returned in buffer. There is no message associated with this condition.
04	83000801	No section names exist. No data was returned.
08	83000750	The buffer is not large enough for one record. No data is returned.
04	83000810	Cursor is negative or beyond the end of the specified item. No data was returned.
12	83000102	Workmod is unbound. GETN request rejected.

Parameter list

If your program does not use the IEWBIND macro, place the address of the GETN parameter list in general purpose register 1.

Table 16. GETN parameter list

```
PARMLIST    DS      0F
            DC      A(GETN)               Function code
            DC      A(RETCODE)            Return code
            DC      A(RSNCODE)            Reason code
            DC      A(WORKMOD)            Workmod token
            DC      A(BUFFER)             Data buffer
            DC      A(CURSOR)             Starting position
            DC      A(COUNT)              Data count
            DC      A(TCOUNT)             Total count
            DC      A(NTYPE+X'800000000') Name type to return and end-of-list indicator
GETN        DC      H'60'                 GETN Function code
            DC      H'version'            Interface version number
NTYPE       DC      CL1'C'                'C' = class; 'S' = section
```

IMPORT: Import a function or external variable

IMPORT describes a function or external variable to be imported and the library member where it can be found.

The syntax of the IMPORT call is:

```
[symbol]    IEWBIND    FUNC=IMPORT

                       [,VERSION=version]
                       [,RETCODE=retcode]
                       [,RSNCODE=rsncode]
                       ,WORKMOD=workmod
                       ,ITYPE=itype
                       ,DLLNAME=dllname
                       ,INAME=iname
                       [,OFFSET=offset]
```

FUNC=IMPORT
Requests import of a function or external variable.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

Note: If Version = 1, 2, or 3 is specified for the IMPORT call, OFFSET cannot be specified as a macro keyword. The parameter list ends with the XINAME parameter (with the high-order bit set).

RETCODE=*retcode* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **– RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that is to receive the reason code returned by the binder.

WORKMOD=*workmod* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the workmod token returned on the CREATEW request. This token must not be changed.

ITYPE=CODE | DATA | CODE64 | DATA64
Specifies whether the imported symbol represents code (a function entry point, for example) or a data item. **CODE64** and **DATA64** are used in conjuction with 64-bit addressing.

DLLNAME=*dllname* **– RX-type address or register (2-12)**
Specifies the name of an area containing a halfword length field followed by the member or alias name of the module containing the imported function or variable. The length field defines the number of characters in the member name and must not be larger than 8 bytes for a directory member or alias, or 1024 bytes for a z/OS UNIX System Services filename.

INAME=*iname* **– RX-type address or register (2-12)**
Specifies the name of an area containing a halfword length field followed by the name of the symbol to be processed. The length field defines the number of characters in the symbol name and must not be larger than 32767.

OFFSET=*offset* **– RX-type address or register (2-12)**
Specifies the name of an area containing a fullword integer. This value is not interpreted by the binder, but will be stored in the import-export table entry for

the symbol if the import is for CODE and it is determined that the symbol specified on the IMPORT statement is to be dynamically resolved.

Processing notes

If DLLNAME was not specified, the IMPORT statement will be ignored. Otherwise, if the symbol is unresolved at the end of autocall and all references have SCOPE=X, the IMPORT request will be converted to an entry in binder class B_IMPEXP. A bind job for a DLL application should include an IMPORT control statement for any DLLs that application expects to use. Otherwise if the DLL name is unresolved at static bind time it will not be accessible at run time (cannot be loaded).

Typically, a library of DLLs has an associated side file of IMPORT control statements, and you can include this side file when statically binding a module that imports functions or variables from that library. You can also edit the records in the side file or substitute your own IMPORT control statements so that some symbols are imported from DLLs in a different library.

Return and reason codes

The common binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion. The import request has been added to the binder's Import/Export list successfully.

Parameter list

If your program does not use the IEWBIND macro, place the address of the IMPORT parameter list in general purpose register 1.

Table 17. IMPORT parameter list

```
PARMLIST    DS      0F
            DC      A(IMPORT)             Function code
            DC      A(RETCODE)            Return code
            DC      A(RSNCODE)            Reason code
            DC      A(WORKMOD)            Workmod token
            DC      A(ITYPE)              Import type
            DC      A(DLLNAME)            DLL Name
            DC      A(INAME)              Section name
            DC      A(OFFSET)             Function descriptor offset
IMPORT      DC      H'38'                 IMPORT function code
            DC      H'4'                  Interface version number
ITYPE       DC      CL1'C'                Import Type

                                            'C' = Code
                                            'D' = Data
                                            'E' = Code64
                                            'F' = Data64
DLLNAME     DC      H'nnn',CLnnn'dllname' DLL containing symbol
INAME       DC      H'nnn',CLnnn'iname'   Imported function or variable
```

Note: The OFFSET parameter is an addition for Version 4, and X'80000000' must be added to either the NAME parameter (for Version 1 through 3) or the OFFSET parameter (for Version 5).

INCLUDE: Include module

INCLUDE brings data into the workmod. The source is usually a data set that can contain a program object, a load module, or a combination of object modules and control statements. In some cases, the program module might be included from virtual storage rather than an external data set. Multiple INCLUDE calls cause all included program objects, load modules, and object modules to be merged in the specified workmod. You can specify a z/OS UNIX System Services file as the DDNAME parameter value on an INCLUDE statement.

The syntax of the INCLUDE call is:

```
[symbol]    IEWBIND   FUNC=INCLUDE

                      [,VERSION=version]
                      [,RETCODE=retcode]
                      [,RSNCODE=rsncode]
                      ,WORKMOD=workmod
                      ,{INTYPE={DEPTR|SMDE},DDNAME=ddname,DEPTR=deptr
                      | INTYPE=NAME,{DDNAME=ddname[,MEMBER=member]
                                    |PATHNAME=pathname}
                      | INTYPE=POINTER,DCBPTR=dcbptr,DEPTR=deptr
                      | INTYPE=TOKEN,EPTOKEN=eptoken}
                      [,ATTRIB={YES | NO}]
                      [,ALIASES={YES | NO}]
                      [,IMPORTS={YES | NO}]
```

FUNC=INCLUDE
Specifies the source of modules to be included in a workmod.

VERSION= *1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

RETCODE=*retcode* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **– RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that is to receive the reason code returned by the binder.

WORKMOD=*workmod* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the workmod token for this request.

INTYPE={NAME | DEPTR | SMDE | POINTER | TOKEN}
Specifies whether the input is identified by a name, a PDS2 directory entry pointer, an SMDE format directory entry pointer, a DCB pointer, or an entry point token.

The parameters applicable for each **INTYPE** are:

INTYPE=DEPTR
DDNAME=*ddname*
DEPTR=*deptr*

INTYPE=NAME
DDNAME=*ddname*
MEMBER=*memberIO*
PATHNAME=*pathname*

INTYPE=POINTER
DCBPTR=*dcbptr*
DEPTR=*deptr*
INTYPE=SMDE
DDNAME=*ddname*
DEPTR=*deptr*
INTYPE=TOKEN
EPTOKEN=*eptoken*

The values are as follows:

NAME
The input is obtained from a sequential data set, a program library, or a z/OS UNIX System Services file.

The DDNAME-MEMBER parameter combination, or PATHNAME, must be specified when INTYPE=NAME.

- If DDNAME refers to a program library, MEMBER must also be specified. NAME is required when **INTENT=BIND**.
- If PATHNAME is specified, it must be an absolute or relative z/OS UNIX System Services pathname that resolves to the desired file (member) name. Note that MEMBER cannot be specified with PATHNAME.

DEPTR
The input is accessed using DDNAME and DEPTR. The directory entry is in PDS2 format.

SMDE
The input is accessed using the DDNAME and DEPTR. The directory entry is in SMDE format.

POINTER
The input is obtained from a member in a partitioned data set. DCBPTR and DEPTR must both be specified. This value is only valid when the processing intent specified on the **CREATEW** call is **ACCESS**.

TOKEN
The input is represented by a token from the CSVQUERY macro. **EPTOKEN** must be specified. This value is only valid when the processing intent specified on the **CREATEW** call is **ACCESS**. The program module has already been loaded into virtual storage. Use this option instead of **POINTER** for modules in PDSE libraries.

The values for INTYPE can be abbreviated as N, D, S, P, or T .

DDNAME=*ddname* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte varying character string that contains the ddname associated with the sequential data set or program library to be included in the workmod. If a program library is specified, MEMBER must also be specified. The DDNAME-MEMBER parameter combination is mutually exclusive with PATHNAME.

Notes:

1. The binder supports all data sets allocated in the extended addressing space (EAS) of an extended address volume (EAV).
2. The binder supports the following dynamic allocation (DYNALLOC or SVC 99) options for all data sets: S99TIOEX(XTIOT), S99ACUCB (NOCAPTURE), and S99DSABA (DSAB above the line).

MEMBER=*member* – RX-type address or register (2-12)
Specifies the location of an 8-byte varying character string that contains the member name or alias of the library member to be included in the workmod.

PATHNAME=*pathname* – RX-type address or register (2-12)
Specifies the location of a 1023-byte varying character string that contains the absolute or relative path name of a z/OS UNIX System Services file. The path name must begin with "/" (absolute path) or "./" (relative path) and is limited to a maximum of 1023 characters. Note that PATHNAME must resolve to the file that is included. PATHNAME is mutually exclusive with the DDNAME-MEMBER parameter combination.

DCBPTR=*dcbptr* – RX-type address or register (2-12)
Specifies the location of a 4-byte pointer that contains the address of a DCB for the partitioned data set or PDSE containing the input module to be included. You code the DCB with the parameters DSORG=PO, MACRF=R, and RECFM=U | F. The DCB must be opened for input or update before calling the INCLUDE function.

Notes:

1. The binder is sensitive to the state of the DCB pointed to by the DCBPTR. The DCB must not be closed and reopened while the binder accesses the corresponding data set during a dialog. Once it is opened initially for an INCLUDE, it must remain open until after the binder's ENDD call takes place.
2. When you alter your DCB as described above, using RESETW (or DELETEW followed by CREATEW) is not enough to reaccess your data set at a later time during the same binder dialog. This only causes the data set's information to remain with the dialog, and such information is no longer valid once the DCB is closed. An attempt to reuse the altered DCB in the same binder dialog can produce unpredictable results.
3. In addition to a caller-created DCB, the following three specific system-provided DCBs can be used here:
   - A linklist DCB (from CVTLINK or DLCBDCB@)
   - A tasklib DCB (from TCBJLB)
   - An EXEC PGM=*.referback DCB (from the top of the DEB chain)
4. If you haven't set a new DCBE with the EADSCB=OK option, the system assumes that the program cannot handle the track address and issues a new ABEND code.
5. The binder supports all data sets allocated in the extended addressing space (EAS) of an extended address volume (EAV).
6. The binder supports the following dynamic allocation (DYNALLOC or SVC 99) options for all data sets: S99TIOEX(XTIOT), S99ACUCB (NOCAPTURE), and S99DSABA (DSAB above the line).

DEPTR=*deptr* – RX-type address or register (2-12)
Specifies the location of a 4-byte pointer that contains the address of a single directory entry for the partitioned data set or PDSE member to be included. The directory entry can be in the PDS2 format or in the SMDE format. If INTYPE=S, it must be SMDE. If INTYPE=D or P, it must be PDS2. The PDS2 format is returned by BLDL while the SMDE format is returned by DESERV. The DEPTR, when used with DCBPTR, will only locate the first data set in a concatenated DD. INTYPE=D or S must be used to locate other data sets in a concatenated DD. This parameter is required for INTYPEs of D, S, or P. DEPTR is valid only if INTENT=ACCESS.

Note: When specifying a directory entry pointer for a directory entry in PDS2 format, the directory entry must contain the full 72 bytes of data.

EPTOKEN=*eptoken* **— RX-type address or register (2-12)**
Specifies the location of an 8-byte area containing the entry point token received with the CSVQUERY macro. EPTOKEN is required when the program module has already been loaded in virtual storage. EPTOKEN is valid only if INTENT=ACCESS.

The binder can retrieve a module identified by an entry point token only if the module was loaded. EPTOKEN cannot be used to retrieve a module in LPA or LLA. EPTOKEN can be used for programs loaded from the UNIX shell by BPX1LOD.

ATTRIB={YES | NO}
Specifies whether to include the program module attributes with the program module. These attributes override attributes set at the dialog level by SETO or STARTD and any attributes set by prior INCLUDE calls. They do not override attributes set at the workmod level by SETO. The values for ATTRIB can be abbreviated as Y or N.

ALIASES={YES | NO}
Specifies whether to include the program module aliases with the program module. Aliases can be included only if you are including a module from a library. If the ddname specified points to a sequential data set, a z/OS UNIX System Services file, or a specific member of a PDS or PDSE library, aliases cannot be included. The values for ALIASES can be abbreviated as Y or N. NO is the default.

IMPORTS={YES | NO}
Indicates whether or not the import statements are to be included from the input module.

Processing notes

If **ATTRIB=YES**, the following attributes are copied from the input directory: AC, AMODE, DC, OL, REUS, RMODE, SSI, TEST, ENTRY POINT, DYNAM, and MIGRATABLE. If INTENT=ACCESS, the following additional attributes are copied: EDITABLE, EXECUTABLE, OVLY, and PAGE-ALIGNED. You cannot set the EXECUTABLE, MIGRATABLE, and PAGE-ALIGNED attributes with either **SETO** or **STARTD**.

When **INTENT=ACCESS** is specified on **CREATEW**, only one data set can be included in the workmod and the data set must be a program object or load module.

If **INTENT=ACCESS** and **ALIASES=YES**, the aliases and any associated addressing modes are included. If **INTENT=BIND** and **ALIASES=YES**, the aliases are included, but the associated addressing modes are not.

If **INTENT=ACCESS** and **ATTRIB=YES**, the **SIGN** option value is preserved in the included module, which means the signature and the directory bit that indicates the presence of the signature are preserved on an INCLUDE API call.

When **INTENT=BIND**, the ddname can refer to a concatenation of data sets or to a z/OS UNIX System Services file. These data sets must be either all libraries or all sequential data sets. If the ddname refers to a library and the member name is included with the library name, it is processed sequentially and can only be concatenated with other library members or sequential data sets. Each library

member must contain either a single program module or a mixture of object modules and control statements. Sequential data sets can only contain object modules and control statements.

The processing of **INCLUDE** can be modified by the **ALTERW** function. CSECTs and symbols can be replaced or deleted in an included module when specified on earlier **ALTERW** calls or equivalent control statements. The scope of such alterations extends only to the first end-of-module condition encountered in the included file. Additional modules can not be included into the workmod once it has been bound.

Return and reason codes

The common binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion. File included successfully.
04	83000515	Unsupported control statement encountered in included file. File included successfully.
04	83000525	An unusual condition was encountered while processing a REPLACE or CHANGE control statement.
04	83000526	An unusual condition was encountered in an input module, while converting it into workmod format. For example, this reason code will be returned if a two-byte relocatable adcon was seen.
08	83000502	One or more editing requests (delete, change or replace operations) failed during inclusion of the module. The module was included successfully, but some of the requested changes were not made.
08	83000505	The included module was marked *not editable*, and has been deleted.
08	83000507	A format error has been encountered in a module being included. The module was not added to the target workmod.
08	83000510	Errors were encountered in the included module. The module is rejected.
08	83000511	A control statement in an included file attempted to include the file containing the statement, or include another file that, in turn, included the original file. The recursive include has been rejected.
08	83000514	The requested member could not be found in the library, or the library could not be found. Request rejected.
08	83000516	A format error has been encountered in one or more control statements being included. The erroneous statements have been ignored.
08	83000517	A NAME control statement was encountered, but no target library (MODLIB) had been specified. The statement was ignored.
08	83000518	A NAME control statement was encountered in a secondary input file. The statement was ignored.
08	83000519	Errors (invalid data) were found in a module being brought in by an INCLUDE control statement. The module was not included.

Return Code	Reason Code	Explanation
08	83000520	The data set or library member specified by an INCLUDE control statement could not be found. The data set or library member was not included.
08	83000521	An I/O error occurred while trying to read an input data set (or directory) specified on an INCLUDE control statement. The data set (or member) was not included.
08	83000522	The input data set specified on an INCLUDE control statement could not be opened. The data set (or member) was not included.
08	83000527	Identify data could not be processed because the section was not included prior to identify statement.
08	83000528	The DE parameter value; the TTR or K (concatenation) field is invalid.
12	83000101	Not all the parameters required for the specified INTYPE (as described above) were provided. The request has been rejected.
12	83000103	The workmod was specified with INTENT=BIND, but the INTYPE was other than DDNAME. The request has been rejected.
12	83000500	The INCLUDE call has attempted to include a second module when the processing intent is ACCESS. The request has been rejected.
12	83000503	An I/O error occurred while trying to read the input data set or its directory. The input is not usable.
12	83000504	The module was successfully included, but the ALIASES or ATTRIB option could not be honored because the directory was not accessible.
12	83000506	An attempt has been made to include an object module into a workmod specified as INTENT=ACCESS. Request rejected.
12	83000507	A format error has been encountered in a module being included. The module was not added to the target workmod.
12	83000509	An attempt has been made to include a file containing control statements but the workmod specified INTENT=ACCESS. The request has been rejected.
12	83000512	The designated source for the current INCLUDE contained more than one module but the target workmod was specified with INTENT=ACCESS. The request has been rejected.
12	83000513	The file could not be opened. Request rejected.
12	83000523	For intent access, the requested module contained a format error and has not been placed in workmod. Request rejected.

If the INCLUDE brings in control statements, the processing of these control statements might generate calls to other binder functions. The errors and their corresponding reason codes from the functions invoked by the generated calls are propagated back to the caller of the INCLUDE function. The functions can include:

- ADDA
- ALIGNT

- ALTERW
- BINDW
- CREATEW
- DELETW
- INSERTS
- ORDERS
- PUTD
- RESETW
- SAVEW
- SETL
- SETO
- STARTS

Parameter list

If your program does not use the IEWBIND macro, place the address of the INCLUDE parameter list in general purpose register 1.

Table 18. INCLUDE parameter list

```
PARMLIST    DS      0F
            DC      A(INCLUDE)              Function code
            DC      A(RETCODE)              Return code
            DC      A(RSNCODE)              Reason code
            DC      A(WORKMOD)              Workmod token
            DC      A(INTYPE)               Intype
            DC      A(DDNAME)               ddname or pathname. Pathname is only valid
                                            if INTYPE=N.
            DC      A(MEMBER)               Member name. A(0) should be coded if
                                            PATHNAME is specified.
            DC      A(DCBPTR)               Pointer to DCB
            DC      A(DEPTR)                Pointer to BLDL entry
            DC      A(EPTOKEN)              EPTOKEN
            DC      A(0)                    Reserved
            DC      A(ATTRIB)               ATTRIB option
            DC      A(ALIASES)              ALIASES option
            DC      A(IMPORTS+X'80000000')  IMPORTS option and end-of-list indicator
INCLUDE     DC      H'40'                   Function code
            DC      H'version'              Interface version number
INTYPE      DC      CL1'N'                  INTYPE source option

                                              'N' = Name
                                              'P' = Pointer
                                              'T' = Token
ATTRIB      DC      CL1'Y'                  ATTRIB option

                                              'Y' = Yes
                                              'N' = No
ALIASES     DC      CL1'Y'                  ALIASES option

                                              'Y' = Yes
                                              'N' = No
IMPORTS     DC      C'Y'                    Whether imports in input should be included

                                              'Y' = Yes
                                              'N' = No
```

Note: X'80000000' must be added to the ALIASES parameter (for Version 1 through 4) and to the IMPORTS parameter (for Version 5).

INSERTS: Insert section

INSERTS positions a control section or named common area within the program module or within overlay segments in an overlay structure. This specification is overridden by the order specified on an ORDERS call.

The syntax of the INSERTS call is:

```
[symbol]    IEWBIND    FUNC=INSERTS

                       [,VERSION=version]
                       [,RETCODE=retcode]
                       [,RSNCODE=rsncode]
                       ,WORKMOD=workmod
                       ,SECTION=section
```

FUNC=INSERTS
Requests that a specified control section be positioned at the current location within the program module or overlay segment.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

RETCODE=*retcode* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **– RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that is to receive the reason code returned by the binder.

WORKMOD=*workmod* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the workmod token for this request.

SECTION=*section* **– RX-type address or register (2-12)**
Specifies the location of a 16-byte varying character string that contains the name of the control section or named common area to be inserted at the current location.

Processing notes

The INSERTS function is valid only when the processing intent is BIND.

In an overlay structure, INSERTS places the section within the overlay segment defined by the preceding OVERLAY control statement or STARTS function. Sections named on insert functions that precede the first STARTS, as well as those not named on any insert statements, are placed in the root segment.

Return and reason codes

The common binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion. Positioning of section will take place during bind operation.
04	83000711	An insert was already processed for this section, and has been replaced.

Return Code	Reason Code	Explanation
12	83000104	INSERT is not valid against a workmod specified with INTENT=ACCESS. Request rejected.

Parameter list

If your program does not use the IEWBIND macro, place the address of the INSERTS parameter list in general purpose register 1.

Table 19. INSERTS parameter list

```
PARMLIST    DS      0F
            DC      A(INSERTS)              Function code
            DC      A(RETCODE)              Return code
            DC      A(RSNCODE)              Reason code
            DC      A(WORKMOD)              Workmod token
            DC      A(SECTION+X'80000000')  Section name and end-of-list indicator
INSERTS     DC      H'36'                   INSERTS function code
            DC      H'version'              Interface version number
```

LOADW: Load workmod

LOADW produces an executable copy of the workmod and returns its entry point and optionally its load point and length. The workmod is bound but control is not passed to it after it is loaded.

Program modules that are identified to the system with this call can later be invoked using the LINK, ATTACH, and XCTL macros. Programs that have not been identified to the system can later be invoked using the CALL macro. See *z/OS MVS Programming: Assembler Services Guide* for information about using these macros.

The syntax of the LOADW call is:

```
[symbol]    IEWBIND    FUNC=LOADW

                       [,VERSION=version]
                       [,RETCODE=retcode]
                       [,RSNCODE=rsncode]
                       ,WORKMOD=workmod
                       ,IDENTIFY={YES | NO}
                       ,EPLOC=eploc
                       [,XTLST=xtlst]
                       [,LNAME=name]
```

FUNC=LOADW
Specifies that the workmod is bound and loaded but not executed.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

RETCODE=*retcode* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that receives the return code returned by the binder.

RSNCODE=*rsncode* **– RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that receives the reason code returned by the binder.

WORKMOD=*workmod* **— RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the workmod token for this request.

IDENTIFY={YES | NO}
Specifies whether or not the loaded program module is identified to the system. If you specify YES, you can optionally provide the name by which the module is to be identified as the argument LNAME. In either case, the entry point address is returned as the argument EPLOC. The value for IDENTIFY can be abbreviated as **Y** or **N**.

EPLOC=*eploc* **— RX-type address or register (2-12)**
Specifies the location of a 4-byte area that receives the entry point address of the loaded program module. This argument is required.

XTLST=*xtlst* **— RX-type address or register (2-12)**
Specifies the address of a buffer that receives the extent list of the loaded program module. This list contains one entry for each contiguous block of storage used for the loaded program module. See Appendix D, “Binder API buffer formats,” on page 239 for a description of the structure of an extent list. This argument is required if you code IDENTIFY=NO; it is optional if you code IDENTIFY=YES.

LNAME=*name* **— RX-type address or register (2-12)**
Specifies the location of an optional 8-byte varying character string that contains the name by which the program module is known to the system. This argument is recognized only when IDENTIFY=YES. If IDENTIFY=YES and this argument is not specified or a null value is provided, the name specified with the LNAME option on a SETO call is used. If no value was specified on a SETO call, the LNAME name defaults to **GO.

Processing notes

Storage for the program module is obtained from the caller's subpool zero. If you code IDENTIFY=NO, the storage described by the extent list should be freed when the program module is no longer required.

If the bound module contains more than one text class, all such classes are concatenated and loaded into contiguous storage locations.

Return and reason codes

The common binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion.
04	83000655	The buffer provided room for only one extent, but a second extent exists for the loaded module. The module was loaded successfully.
04	83000603	The AMODE or RMODE of one or more input ESD records is incompatible with the AMODE or RMODE of the primary entry point.
04	83000604	There was a conflict in the AMODE/RMODE specification of the current module. This means that 1) The AMODE/RMODE combination is invalid, or one of the MODEs is invalid, or 2) OVLY was specified but either AMODE or RMODE is not (24). The module was loaded with AMODE(24) and RMODE(24).

Return Code	Reason Code	Explanation
04	83000605	No entry name has been provided, either by the user or from any object module processed. The entry point will default to the first text byte.
04	83000607	The module was loaded successfully, but the indicated 2-byte adcon(s) did not relocate correctly.
04	83000657	The module was loaded with AMODE(24), but one or more references in the module were resolved to modules in the Extended LPA. Load successful.
08	83000306	The module was loaded, but the binder could not produce the load summary report.
08	83000650	The entry name specified was not defined in the loaded module. The entry point was forced to the first text byte.
12	83000101	Identify was set to NO, but no extent list buffer was provided. Request rejected.
12	83000415	The module to be loaded contains no text. Execution impossible.
12	83000651	The IDENTIFY for the loaded module failed, probably due to the existence of another module of the same name. The module was loaded successfully, but cannot be accessed by system-assisted linkage.
12	83000652	Sufficient storage was not available to load the module. The module is not loaded.
12	83000653	An error of severity greater than that allowed by the current LET value was encountered. The module is not loaded.
12	83000656	The module was bound in overlay format, and cannot be loaded. Request rejected.

Parameter list

If your program does not use the IEWBIND macro, place the address of the LOADW parameter list in general purpose register 1.

Table 20. LOADW parameter list

```
PARMLIST    DS      0F
            DC      A(LOADW)              Function code
            DC      A(RETCODE)            Return code
            DC      A(RSNCODE)            Reason code
            DC      A(WORKMOD)            Workmod token
            DC      A(IDENT)              Identify option
            DC      A(EPLOC)              Entry point address
            DC      A(XTLST)              Extent list
            DC      A(NAME+X'80000000')   Name used for identify and end-of-list indicator
LOADW       DC      H'81'                 LOADW function code
            DC      H'version'            Interface version number
IDENT       DC      CL1'Y'                Identify option

                                            'Y' = Yes
                                            'N' = No
```

ORDERS: Order sections

ORDERS allows you to specify the location of a section (control section or common area) within the program module. You determine the sequencing of multiple sections using multiple ORDERS requests.

The syntax of the ORDERS call is:

```
[symbol]    IEWBIND    FUNC=ORDERS

                       [,VERSION=version]
                       [,RETCODE=retcode]
                       [,RSNCODE=rsncode]
                       ,WORKMOD=workmod
                       ,SECTION=section
```

FUNC=ORDERS
Requests the order section function. It can be abbreviated as ORDER.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

RETCODE=*retcode* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **– RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that is to receive the reason code returned by the binder.

WORKMOD=*workmod* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the workmod token for this request.

SECTION=*section* **– RX-type address or register (2-12)**
Specifies the location of a 16-byte varying character string that contains the name of the section to be ordered.

Processing notes

An ORDERS request is valid only when the processing intent is BIND.

ORDERS requests cause the named sections to be moved to the beginning of the module or overlay segment in the same sequence as the ORDERS calls are received by the binder. If a section name appears in more than one call, the last request is used.

Reordering does not occur until the workmod is bound, regardless of when the ORDERS call is made.

Sections that are not specified in ORDERS calls follow sections that have been ordered. In an overlay module, ORDERS calls can specify sections in more than one segment but sections will never be moved from one segment to another.

Return and reason codes

The common binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion. Sections will be ordered during bind processing.
08	83000711	A previous order request for this section was received, and has been replaced.
12	83000104	An ORDERS request is invalid against a workmod specified with INTENT=ACCESS. Request rejected.

Parameter list

If your program does not use the IEWBIND macro, place the address of the ORDERS parameter list in general purpose register 1.

Table 21. ORDERS parameter list

```
PARMLIST    DS      0F
            DC      A(ORDERS)               Function code
            DC      A(RETCODE)              Return code
            DC      A(RSNCODE)              Reason code
            DC      A(WORKMOD)              Workmod token
            DC      A(SECTION+X'80000000')  Section name and end-of-list indicator
ORDERS      DC      H'37'                   ORDERS function code
            DC      H'version'              Interface version number
```

PUTD: Put data

PUTD stores data into a new or existing workmod item. If the item already exists, the data overlays existing data in the item, or is added at the end. If the item does not yet exist, a new one is created using the specified class and section names.

The syntax of the PUTD call is:

```
[symbol]    IEWBIND    FUNC=PUTD

                       [,VERSION=version]
                       [,RETCODE=retcode]
                       [,RSNCODE=rsncode]
                       ,WORKMOD=workmod
                       ,CLASS=class
                       ,SECTION=section
                       [,AREA=buffer
                       [,CURSOR=cursor]
                       [,COUNT=count]
                       [,NEWSECT={NO | YES}]
                       [,ENDDATA={NO | YES}]
```

label
Optional symbol. If present, the label must begin in column 1.

FUNC=PUTData
Requests the Put Data function. It can be truncated to PUTD.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

RETCODE=*retcode* **– RX-type address or register (2-12)**
Specifies the name of a fullword integer variable that is to receive the completion code returned by the binder.

RSNCODE=*rsncode* **– RX-type address or register (2-12)**
Specifies the name of a 4-byte hexadecimal string variable that is to receive the reason code returned by the binder.

WORKMOD=*workmod* **– RX-type address or register (2-12)**
Specifies the name of an 8-byte variable that contains the Workmod Token returned by the program management binder on the CREATEW request. This token must not be modified by the caller.

CLASS=*class* **– RX-type address or register (2-12)**
Specifies the name of an 16-byte varying character string variable containing the class name, left adjusted.

Certain binder-defined classes are generated by the binder and should not be specified if the workmod was created with INTENT=BIND

B_PRV	Pseudoregister vector
B_IDRB	Binder identification record
B_MAP	Module map
B_PARTINIT	Part initializers
B-IMPEXP	Import export table
B_LIT	Loader information table

SECTION=*section* **– RX-type address or register (2-12)**
Specifies the name of a 32K–1 byte varying character string variable containing the name of the section to be processed.

AREA=*buffer* **– RX-type address or register (2-12)**
Specifies the name of a standard buffer containing the data to be replaced in or appended to the designated workmod item. This parameter is not required if the caller wants only to signal an end-of-section condition. When AREA is used, COUNT also must be specified.

CURSOR=*cursor* **– RX-type address or register (2-12)**
Specifies the name of a fullword integer variable that indicates to the program management binder the position (relative record or byte) in the item at which to store the buffered data.

If CURSOR= is omitted or CURSOR= -1, the buffered data is appended to the existing item. If INTENT=ACCESS, only B_IDR- and B_SYM-class items can be appended. CURSOR is valid only if AREA has been specified.

COUNT=*count* **– RX-type address or register (2-12)**
Specifies the name of a fullword integer variable containing the number of data bytes or records to be inserted. COUNT cannot be specified unless AREA is specified.

NEWSECT={NO | YES}
Specifies that one or more new sections are being added to the workmod, and that the data present in the buffer belongs to one of those sections. Once NEWSECT=YES has been specified, all subsequent PUTD calls must also indicate NEWSECT=YES, until all of the new sections are complete (see ENDDATA parameter, below). YES or NO can be abbreviated Y or N, respectively. NO is the default.

ENDDATA={NO | YES}
ENDDATA=YES indicates that all of the sections being added by this series of PUTD calls are complete, and that certain validity checks are to be performed on the ESD and RLD. Any data in the buffer is added to the workmod before validation begins. YES or NO can be abbreviated Y or N, respectively. NO is the default.

Processing notes

CLASS can contain any valid data class except B_IDRB, and SECTION should contain the name of the CSECT being created or updated. If INTENT=ACCESS, ESD and RLD data cannot be modified, nor TEXT extended.

CURSOR allows the caller to replace part of an existing item. It contains the offset, relative to the start of the item, where the buffered data is to be inserted. If CURSOR = is omitted or CURSOR contains -1, the buffered data is added to the end of the item. If CURSOR contains zero, the data is stored starting at the first byte or record.

The BUFFER must be in the standard format for the data class being stored. See Appendix D, “Binder API buffer formats,” on page 239 for additional information on standard buffer formats.

The binder moves the specified number of bytes or records from the buffer into workmod. Data is reformatted, if necessary, to conform to the internal format of the data in workmod.

PUTD operates in either INPUT or EDIT mode. INPUT mode is used when adding new sections to the workmod (INTENT must be BIND), and begins with the first PUTD call specifying NEWSECT=YES and continues until ENDDATA=YES is received. While in INPUT mode, there are certain restrictions on acceptable program management binder functions. The only functions allowed against a workmod in INPUT mode (those for which an input workmod token matches that of the workmod in INPUT mode) are PUTD with NEWSECT = YES, RESETW, or DELETEW. RESETW, DELETEW, and ENDD causes the operation to be prematurely terminated.

In INPUT mode, sections being added are held in a temporary workmod until ENDDATA=YES is received, when all of the new sections are validated as a unit and added to the target workmod. If any of the new sections fail validation, the entire group is discarded; otherwise, sections are added to the permanent workmod according to normal merge rules. If the new section already exists, it does not replace the existing one. If a deferred ALTERW request is pending, it is applied to all sections in the temporary workmod before merging them into the permanent workmod.

Certain additional requirements are placed on the user when entering module data in input mode.

- Only one private code and/or blank common section can be handled during one PUTD call series (a series being terminated by an end-of-data indication).
- For any section, the first class received must be B_ESD. More than one section can be passed in a PUTD series, but the first class in any section must be B_ESD.
- The first record in the first buffer of any B_ESD element must be the section definition record (SD). This ensures that the first PUTD call for any section identifies the section type.

Validation of the module in the temporary workmod proceeds one section at a time. Violation of any of the following restrictions causes a validation failure for that section:

- No ESD item exists for the section.
- The ESD item does exist, but does not contain a type SD ESD record.
- One or more of the ESD records contains an invalid ESD_TYPE. Only types SD, CM, ED, LD, ER, and PR are expected via PUTD. ESD_TYPEs of ST, ET, DS and PD are not acceptable for PUTD input.
- One or more LD records have a section offset (ESD_SECTION_OFFSET) greater than the ESD section length (ESD_LENG from the SD record).
- Text length exceeds ESD section length.
- (RLD_SECTION_OFFSET + RLD_ADCON_LENGTH) exceeds ESD section length.

EDIT mode is used to update existing data items, or to add new items to an existing section. EDIT mode begins with the first PUTD call specifying NEWSECT=NO (the default) and continues until ENDDATA=YES is received. In EDIT mode, each PUTD call is completely processed before returning to the caller. Some validation is performed on ESD and RLD type data as it is received, to prevent consistency or integrity problems in the target module. This includes all of the same checks listed above for input mode, except the last.

Note: Because EDIT mode checking is done on a single buffer rather than an entire item, the sequence in which the individual data classes are updated can affect the successful validation of the buffered data. To avoid possible timing problems, section data should be updated in the following sequence:
- ESD section record (type SD or CM)
- Other ESD data
- Text
- RLD
- Other classes

If INTENT=ACCESS, certain restrictions apply. B_ESD and B_RLD items cannot be created or modified. TEXT items cannot be created or have their lengths extended.

If INTENT=BIND, SECTION must contain blanks or a user-assigned name consisting only of characters between X'41' and X'FE' (or X'0E' or X'0F'). A section name of all blanks should be used for private code or blank common. Special names, which always begin with a character invalid for user-assigned names, are created by the program management binder and can only be used by the caller when modifying an existing item. Such modification requires INTENT=ACCESS. Private code special names created by the program management binder can not be used in the PUTD call for section name with INTENT=BIND.

Only one private code or blank common section (not both) can be handled during one PUTD call series (a series being terminated by an end-of-data indication). Section names used in the SECTION field of the call parameter list must be the same as the ESD_NAME field in the supplied ESD input record.

Return and reason codes

The common binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion. Data inserted in workmod.

Return Code	Reason Code	Explanation
04	83000801	No data was passed. Module has not been changed.
08	83000815	Errors (such as invalid names) found in records contained in PUTD data buffers. Some data might have been dropped.
12	83000101	Buffer not large enough to contain the designated number of bytes or records. Request rejected.
12	83000802	NEWSECT was set to 'Yes', but workmod intent is ACCESS. Request rejected.
12	83000803	Workmod intent is ACCESS, but the target section does not exist. Request rejected.
12	83000804	PUTD cannot be used to modify ESD or RLD data (even in an existing section) in a workmod specified with INTENT=ACCESS. Request rejected.
12	83000805	PUTD cannot be used to extend the length of text data in a workmod specified with INTENT=ACCESS. Request rejected.
12	83000806	PUTD cannot be used to modify sections generated by the binder. Request rejected.
12	83000807	Incorrect parameter specification. The call attempted to modify an existing item with NEWSECT=YES, or it specified NEWSECT=NO while the binder was still in input mode. Request rejected.
12	83000808	PUTD cannot be used to modify the IDRB record. Request rejected.
12	83000811	One or more errors was detected in the module just completed. The CSECT(s) was not added to the workmod.
12	83000814	One or more errors were detected in the data records in the buffer just passed to the program management binder. The records were not added to workmod.

Parameter list

Table 22. PUTD parameter list

```
PARMLIST    DS      0F
            DC      A(PUTD)                 Function code
            DC      A(RETCODE)              Return code
            DC      A(RSNCODE)              Reason code
            DC      A(WORKMOD)              Workmod token
            DC      A(CLASS)                Class name
            DC      A(SECTION)              Section name
            DC      A(BUFFER)               Standard data buffer
            DC      A(CURSOR)               Starting position
                                            To append data, set cursor value to (-1).
            DC      A(COUNT)                Data count
            DC      A(NEWSECT)              New section flag
            DC      A(ENDDATA+X'80000000')  End-of-data flag
PUTD        DC      H'65'                   PUTD function code
            DC      H'version'              Interface version number
```

Table 22. PUTD parameter list (continued)

NEWSECT	DC	CL1'Y'	New section flag 'Y' = Yes 'N' = No
ENDDATA	DC	CL1'Y'	End-of-data flag 'Y' = Yes 'N' = No

RENAME: Rename symbolic references

This function renames a symbolic reference if a reference to the old name remains unresolved at the end of the first pass of autocall. For a complete explanation of the autocall process, see the information on resolving external references in *z/OS MVS Program Management: User's Guide and Reference*.

The syntax of the RENAME call is:

```
[symbol]    IEWBIND    FUNC=RENAME

                       [,VERSION=version]
                       [,RETCODE=retcode]
                       [,RSNCODE=rsncode]
                       ,WORKMOD=workmod
                       ,OLDNAME=oldname
                       ,NEWNAME=newname
```

FUNC=RENAME
Requests the RENAME function.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

RETCODE=*retcode* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **– RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that is to receive the reason code returned by the binder.

WORKMOD=*workmod* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the workmod token returned on the CREATEW request. You must not modify this token.

OLDNAME=*oldname* **– RX-type address or register (2-12)**
Specifies the location of a 32K–1 byte varying character string that contains the symbol to be renamed. The format consists of a 2-byte length field followed by the actual name. The length does not include the first two bytes.

NEWNAME=*newname* **– RX-type address or register (2-12)**
Specifies the location of a 32K–1 byte varying character string that contains the symbol to which the old name should be changed. The format consists of a 2-byte length field followed by the actual name. The length does not include the first two bytes.

Processing notes

The only immediate result of the RENAME API call is that the new rename request will be added to the list of such requests. Nothing else will be done until final autocall processing. At the end of the first pass of autocall (that is, when all possible references have been resolved with the names as they were on input), rename processing will be performed.

Return and reason codes

The common binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion. The binder added the rename request to the rename list successfully.
04	83000501	Either the OLDNAME or the NEWNAME already existed in the binder rename list. The rename request was not successful (i.e., the request was not added to the binder rename list).

Parameter list

If your program does not use the IEWBIND macro, place the address of the RENAME parameter list in general purpose register 1.

Table 23. RENAME parameter list

```
PARMLIST    DS      0F
            DC      A(RENAME)               Function code
            DC      A(RETCODE)              Return code
            DC      A(RSNCODE)              Reason code
            DC      A(WORKMOD)              Workmod token
            DC      A(OLDNAME)              Reason code
            DC      A(NEWNAME+X'80000000')  RENAME list & end-of-list indicator
RENAME      DC      H'22'                   RENAME function code value
            DC      H'version'              Interface version number
OLDNAME     DC      H'nnn',CLnnn            Old name
NEWNAME     DC      H'nnn',CLnnn            New name
```

RESETW: Reset workmod

RESETW resets a workmod to its initial state. All items are deleted. Options are reset to the options current for the dialog and the processing intent must be respecified. The workmod token is not changed.

If a workmod has been changed without being saved or loaded, it cannot be reset without specifying PROTECT=NO.

The syntax of the RESETW call is:

```
[symbol]        IEWBIND     FUNC=RESETW

                            [,VERSION=version]
                            [,RETCODE=retcode]
                            [,RSNCODE=rsncode]
                            ,WORKMOD=workmod
                            ,INTENT={BIND | ACCESS}
                            [,PROTECT={YES | NO}]
```

FUNC=RESETW
Specifies that a workmod be reset to its original state.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

RETCODE=*retcode* **— RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **— RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that is to receive the reason code returned by the binder

WORKMOD=*workmod* **— RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the workmod token for this request.

INTENT={BIND | ACCESS}
Specifies the range of binder services that can be requested for this workmod. The possible arguments are as follows:

BIND
Specifies that the processing intent for this workmod is bind. The workmod will be bound, and all binder functions can be requested.

ACCESS
Specifies that the processing intent for this workmod is access. The workmod will not be bound, and no services that alter the size or structure of the program module can be requested. See “Processing intents” on page 10 for a list of the services that are not allowable.

The argument for INTENT can be abbreviated as **B** or **A**.

PROTECT={YES | NO}
Specifying **PROTECT=NO** allows the binder to reset a workmod that has been altered but not yet saved or loaded. The argument for PROTECT can be abbreviated as **Y** or **N**. YES is the default.

Processing notes

The binder is sensitive to the state of the DCB pointed to by the DCBPTR in an INCLUDE call. The DCB must not be closed and reopened while the binder accesses the corresponding data set during a dialog. Once it is opened initially for an INCLUDE call, it must remain open until after the binder's ENDD call takes place.

Note that if you do alter your DCB as described above, using RESETW (or DELETEW followed by CREATEW) is not enough to reaccess your data set at a later time during the same binder dialog. This only causes the data set's information to remain with the dialog, and such information is no longer valid once the DCB is closed. An attempt to reuse the altered DCB in the same binder dialog might produce unpredictable results. To avoid this, end your dialog (ENDD) and start a new one (STARTD).

Return and reason codes

The common binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion. Workmod has been reset.

Return Code	Reason Code	Explanation
08	83002624	Unexpected error from SIGCLEAN.
12	83000709	The workmod was in an altered state, but PROTECT=YES was specified or defaulted. RESETW request rejected.

Parameter list

If your program does not use the IEWBIND macro, place the address of the RESETW parameter list in general purpose register 1.

Table 24. RESETW parameter list

```
PARMLIST    DS      0F
            DC      A(RESETW)               Function code
            DC      A(RETCODE)              Return code
            DC      A(RSNCODE)              Reason code
            DC      A(WORKMOD)              Workmod token
            DC      A(INTENT)               Processing intent
            DC      A(PROTECT+X'80000000')  Protection flag and end-of-list indicator
RESETW      DC      H'19'                   RESETW function code
            DC      H'version'              Interface version number
INTENT      DC      CL1'A'                  Processing intent

                                              'A' = Access
                                              'B' = Bind
PROTECT     DC      CL1'Y'                  Protection flag

                                              'Y' = Yes
                                              'N' = No
```

SAVEW: Save workmod

SAVEW saves a workmod either as a load module in a partitioned data set or in a PDSE or a z/OS UNIX System Services file. If the workmod has not already been bound, it is bound before being saved.

The syntax of the SAVEW call is:

```
[symbol]    IEWBIND    FUNC=SAVEW

                       [,VERSION=version]
                       [,RETCODE=retcode]
                       [,RSNCODE=rsncode]
                       ,WORKMOD=workmod
                       [,MODLIBddname | pathname]
                       [,SNAME=member]
                       [,REPLACE={YES | NO}]
```

FUNC=SAVEW
Specifies that a workmod is to be bound and stored in a program library.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

RETCODE=*retcode* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **– RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that is to receive the reason code returned by the binder.

WORKMOD=*workmod* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the workmod token for this request.

MODLIB=*ddname* | *pathname* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte varying character string that contains the DD name of the target library. If this parameter is not specified, the MODLIB value set on the STARTD or SETO call is used.

pathname specifies the location of a 1023-byte varying character string that contains the absolute or relative path name of a z/OS UNIX System Services file. The path name must begin with "/" (absolute path) or "./" (relative path) and is limited to a maximum of 1023 characters. Note that *pathname* must resolve to the file that is saved. *pathname* is mutually exclusive with *ddname*.

SNAME=*member* **– RX-type address or register (2-12)**
Specifies the location of an 1024-byte varying character string that contains the member name of the program to be saved in the target library. If this parameter is not specified, the member name on the DD statement for the program library is used. If neither of these is specified, the SNAME value on the STARTD or SETO call is used. If no value for SNAME is specified anywhere, the call fails. If SNAME exceeds 8 bytes, the binder generates an 8-byte primary name and saves the specified name as a special alias, called the *alternate primary*. For more information on long names see the NAME statement in *z/OS MVS Program Management: User's Guide and Reference*.

REPLACE={YES | NO}
Specifies whether or not the program module will replace an existing member of the same name in the target library. This argument can be abbreviated as **Y** or **N**. NO is the default.

Processing notes

Aliases that have been specified on ADDA calls or included with an input module are added to the library directory. Existing aliases that are not specified on a replacement module are deleted if the target library is a PDSE. They remain unchanged if the target library is a partitioned data set. All aliases should be respecified to ensure proper updating.

If the NE (not editable) option has been specified, or if a module with the NE attribute is copied, no ESD items are saved and the module is marked *not-editable* in the directory entry. Not-editable load modules or PM1 format program objects can be reprocessed by the binder if INTENT=ACCESS. Not-editable PM2 and higher format program objects cannot be reprocessed by the binder, even if INTENT=ACCESS.

If you specify **REPLACE=NO** when processing a z/OS UNIX System Services file, the binder issues an informational message.

If any of the following conditions exist, the output module is not saved:

- The module was bound with the RES option and one or more references were resolved to modules in the link pack area.

- The module was marked not-executable and an executable module of the same name already exists in the target library. This restriction can be overridden through the use of the STORENX option. For more information, see the STORENX option in *z/OS MVS Program Management: User's Guide and Reference*.
- The target library is a partitioned data set and the module exceeds the restrictions for load modules. To overcome this problem, change the target library to a PDSE and save the module as a program object.
- Saving DLL modules and their side files
  - Saving side files

    When modules are enabled for dynamic linking, a side file can be generated to go along with the saved module. The side file contains IMPORT control statements that describe which function and data items to import from which dynamic link libraries in order to resolve references to symbols dynamically. The name of the saved module is also used as the member name for the side file whose ddname is specified in the STARTDialog binder API if the side file was allocated as a library or a z/OS UNIX System Services directory. If the module is saved to a z/OS UNIX System Services file (that is, if SYSLMOD is a z/OS UNIX System Services file), the module name can be up to 255 bytes. However, if the target library for the side file is a PDS or a PDSE and the module name is greater than eight bytes, that name cannot be used for the side file because the maximum member name length for PDS/PDSE data sets is 8 bytes. The side file is not saved in this case. To solve this problem, either shorten the z/OS UNIX System Services member name to 8 bytes or less, or change the side file DDNAME to represent a z/OS UNIX System Services file.
  - Saving DLLs

    While creating a definition side file, the binder uses the module name specified in the NAME control statement or SAVEW API for the DLLNAME parameter of the IMPORT control statements. For more information, see the IMPORT statement in *z/OS MVS Program Management: User's Guide and Reference*. If this is a long name (greater than 8 bytes), the binder generates a unique 8-byte name for the DLL module if the module is saved to a PDS or a PDSE program library. Therefore, any applications using the side file (whose IMPORT control statements reference the long DLL name) will be unable to dynamically link to said DLL because the DLL name will have been modified (shortened). Because of this, long names should not be used for DLLs unless the DLL module is saved to a z/OS UNIX System Services file.

Return and reason codes

The common binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion. Module and aliases saved in designated library.
04	83000403	The reusability of one or more sections was less than that specified for the module.
08	8E000401	One or more SYMLINK requests failed.
04	83000410	An error was encountered while saving a workmod. The module was saved, but might not be executable.
04	83000411	A module saved as a program object had the SCTR attribute specified. The SCTR attribute was ignored.

Return Code	Reason Code	Explanation
04	83000420	A module saved as a load module contained contents not compatible with that format. Some auxiliary information might have been lost (for example, IDRU records may have been truncated or discarded).
04	83000605	No entry name has been provided, either by the user or from any object module processed. The entry point will default to the first text byte.
04	83000606	One or more RMODE(24) sections have been included in an RMODE(ANY) module.
04	83000423	While attempting to process a side file, the binder can issue this code for two reasons: 1) You did not specify a side file ddname in the FILES parameter of the STARD call, or 2) Inside your side file, one of the IMPORT control statements refers to a DLLNAME that is longer than 8 bytes and resides in a PDS or PDSE. If the first reason does not apply to your case, refer to *Saving DLLs* in the processing notes of SAVEW for additional information on the second reason.
04	83000604	There was a conflict in the AMODE/RMODE specification of the current module. This means that 1) The AMODE/RMODE combination is invalid, or one of the MODEs is invalid, or 2) OVLY was specified but either AMODE or RMODE is not (24). In the first case, the module is saved with the MODEs described in the section on AMODE and RMODE validation in *z/OS MVS Program Management: User's Guide and Reference*. In the second case, the module is saved with AMODE(24) and RMODE(24).
04	83000607	The saved program module contains 2-byte adcons that cannot be relocated.
04	83002631	ALIASES=ALL was specified and one or more of the eligible symbols was longer than 1024 bytes. Aliases were not created for these symbols.
08	83000400	The module has been saved as requested, but has been marked *not-editable*.
08	83000401	One or more aliases could not be added to the target directory. Module saved as requested.
08	83000306	The module was saved successfully, but the save operation summary could not be printed.
08	83000402	The entry name specified is not defined in the module being saved. The entry point will default to the first text byte.
08	83000603	The AMODE or RMODE of one or more input ESD records is incompatible with the AMODE or RMODE of the primary entry point.

Return Code	Reason Code	Explanation
08	83000425	A module name exceeds the maximum name length allowed for a member in a side file data set. When modules are enabled for dynamic linking, a side file can be generated to go along with the saved module. The name of the saved module is also used as the name for the side file whose ddname is specified in the STARTD binder API. If the module is saved to a z/OS UNIX System Services file (that is, if SYSLMOD is a z/OS UNIX System Services file), the module name can be up to 255 bytes. However, if the side file is saved to a PDS or a PDSE, that name cannot be used for the side file because the maximum member name length for such data sets is 8 bytes. Either shorten the z/OS UNIX System Services member name to 8 bytes or less, or change the side file DDNAME to represent a z/OS UNIX System Services file.
08	83002622	The signed module is saved to the format that does not support signing.
08	83002623	Unexpected return code from RACF call.
12	83000404	The module exceeded the limitations for load modules, and could not be saved in the specified PDS library.
12	83000405	A permanent write error was encountered while attempting to write the load module. The save operation terminated prematurely, and the module is unusable.
12	83000406	A permanent read error was encountered while attempting to write the load module. The save operation terminated prematurely, and the module is unusable.
12	83000407	No valid member name has been provided. Request rejected.
12	83000408	The workmod has been marked not executable, and cannot replace an executable version. Request rejected.
12	83000409	A member of the same name already exists in the target library, but the REPLACE option was not specified. The module was not saved.
12	83000412	The module contained no text and could not be saved.
12	83000413	One or more external references in the workmod were bound to modules in the Link Pack Area. The module cannot be saved.
12	83000414	The workmod is null. No modules were successfully included from any source file. The workmod cannot be saved.
12	83000415	The module is empty (contains no nonempty sections) and will not be saved unless LET=12.
12	83000416	No ddname has been specified for the target library. Request rejected.
12	83000417	The target data set is not a library. Request rejected.
12	83000418	Target data set of SAVEW does not have a valid record format for a load library. Module not saved. **Note:** This can happen if the output PDS or PDSE has a record format specified other than U.
12	83000421	Text longer than 1 gigabyte in program object. Module not saved.

Return Code	Reason Code	Explanation
12	83000422	The workmod contained data that cannot be saved in the requested format. The module is not saved.
12	83000424	A data management error was encountered while attempting to open, close, read, or write to a definition side file. Module not saved.
12	83000600	The target library could not be found. Request rejected.
12	83000601	The binder could not successfully close the output library.
12	83000602	The binder could not successfully open the output library.

Parameter list

If your program does not use the IEWBIND macro, place the address of the SAVEW parameter list in general purpose register 1.

Table 25. SAVEW parameter list

```
PARMLIST    DS      0F
            DC      A(SAVEW)                Function code
            DC      A(RETCODE)              Return code
            DC      A(RSNCODE)              Reason code
            DC      A(WORKMOD)              Workmod token
            DC      A(DDNAME)               Library ddname
            DC      A(MEMBER)               Library member name
            DC      A(REPLACE+X'80000000')  Replace option and end-of-list indicator
SAVEW       DC      H'80'                   SAVEW function code
            DC      H'version'              Interface version number
REPLACE     DC      CL1'Y'                  Replace option

                                              'Y' = Yes
                                              'N' = No
```

SETL: Set library

SETL specifies how a specified symbol will be handled during automatic library call. SETL is not performed until the workmod is bound, regardless of where the call appears in the dialog.

The syntax of the SETL call is:

```
[symbol]        IEWBIND     FUNC=SETL

                            [,VERSION=version]
                            [,RETCODE=retcode]
                            [,RSNCODE=rsncode]
                            ,WORKMOD=workmod
                            [,SYMBOL=symbol]
                            [,LIBOPT={
                            CALL | NOCALL | EXCLUDE}]
                            [,CALLIB=ddname | ,PATHNAME=pathname]
```

FUNC=SETL

Specifies that you are requesting an automatic library call option for a symbol. The particular library call option is set on the LIBOPT parameter.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

RETCODE=*retcode* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **– RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that is to receive the reason code returned by the binder.

WORKMOD=*workmod* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the workmod token for this request.

SYMBOL=*symbol* **– RX-type address or register (2-12)**
Specifies the location of a 32K–1 varying character string that contains the name of the symbol for which a library call option is being set. This parameter is optional with LIBOPT=CALL. If omitted, the service call is interpreted as a request to allow any accessible symbol in the CALLIB or PATHNAME to be used during final symbol resolution.

LIBOPT={CALL | NOCALL | EXCLUDE}
Requests whether the automatic library call option for a symbol be call, nocall, or exclusive nocall. The possible arguments are as follows:

CALL
Specifies a search of the library specified by CALLIB to resolve the reference. If the symbol cannot be resolved from this library, no attempt to resolve it from the system autocall library is made.

CALL is the default.

NOCALL
Specifies that no attempt is made to resolve the reference via autocall during the current dialog.

EXCLUDE
Specifies that no attempt is made to resolve the reference via autocall during the current dialog or during any subsequent binder processing. This can be overridden in subsequent processing runs by resetting the LIBOPT value to CALL on a SETL call.

The argument for LIBOPT can be abbreviated as **C**, **N**, or **E**.

CALLIB=*ddname* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte varying character string that contains the ddname of the library to be used during autocall. CALLIB is mutually exclusive with PATHNAME. This keyword is only recognized if LIBOPT=CALL is coded or defaulted.

PATHNAME=*pathname* **– RX-type address or register (2-12)**
Specifies the location of a 1023-byte varying character string that contains the absolute or relative path name of a z/OS UNIX System Services file. The path name must begin with "/" (absolute path) or "./" (relative path) and is limited to a maximum of 1023 characters. Note that PATHNAME must resolve to the file that is included. PATHNAME is mutually exclusive with CALLIB. This keyword is only recognized if LIBOPT=CALL is coded or defaulted.

Processing notes

When multiple conflicting SETL requests are made, the last issued is used. If a SETL CALL request is made that does not list a symbol, followed by NOCALL or EXCLUDE requests for the same CALLIB or PATHNAME, other symbols that have not been restricted can still be used from that library or path.

A SETL request is valid only when the processing intent is BIND.

Return and reason codes

The common binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion.
08	83000711	This request replaced a previous SETLIB request for the same symbol.
12	83000101	The LIBOPT and CALLIB parameters are inconsistent. Either LIBOPT=C and CALLIB was omitted, or LIBOPT=N or E and CALLIB was present. Request rejected.
12	83000104	The SETL function is invalid against a workmod specified with INTENT=ACCESS. Request rejected.

Parameter list

If your program does not use the IEWBIND macro, place the address of the SETL parameter list in general purpose register 1.

Table 26. SETL parameter list

```
PARMLIST    DS      0F
            DC      A(SETL)                 Function code
            DC      A(RETCODE)              Return code
            DC      A(RSNCODE)              Reason code
            DC      A(WORKMOD)              Workmod token
            DC      A(SYMBOL)               Symbol/Section name
            DC      A(LIBOPT)               Library option
            DC      A(DDNAME+X'80000000')   Library ddname or pathname
SETL        DC      H'21'                   SETL function code
            DC      H'version'              Interface version number
XLIBOPT     DC      CL1'C'                  Library option

                                              'C' = Call
                                              'N' = Nocall
                                              'E' = Exclude
```

SETO: Set option

SETO specifies options for processing and module attributes. Each option is set at either the dialog or workmod level by providing a token in the call. The options that can be specified are listed in Chapter 7, “Setting options with the regular binder API,” on page 173.

The syntax of the SETO call is:

```
[symbol]    IEWBIND    FUNC=SETO

                       [,VERSION=version]
                       [,RETCODE=retcode]
                       [,RSNCODE=rsncode]
                       [,WORKMOD=workmod]
                       [,DIALOG=dialog]
                       ,OPTION=option
                       ,OPTVAL=optval
                       [,PARMS=parms]
```

FUNC=SETO
Specifies that you are requesting specific processing options or module attributes for a dialog or workmod.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

Note: If VERSION=1 is specified for the SETO call, PARMS cannot be specified as a macro keyword. The parameter list ends with the OPTVAL parameter (with the high-order bit set). This exception is for version 1 only.

RETCODE=*retcode* **— RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **— RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that is to receive the reason code returned by the binder.

WORKMOD=*workmod* **— RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the workmod token for this request. WORKMOD and DIALOG are mutually exclusive. To set the options at the workmod level, provide the WORKMOD token.

DIALOG=*dialog* **— RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the appropriate dialog token. WORKMOD and DIALOG are mutually exclusive. To set the options at the dialog level, provide the DIALOG token.

OPTION=*option* **— RX-type address or register (2-12)**
Specifies the location of an 8-byte varying character string that contains an option keyword. Except for CALLIB, all keywords can be truncated to three characters. See Chapter 7, “Setting options with the regular binder API,” on page 173 for a complete list of keywords.

OPTVAL=*optval* **— RX-type address or register (2-12)**
Specifies the location of a varying character string that contains a value or a list of values for the specified option.

PARMS=*parms* **— RX-type address or register (2-12)**
Specifies the location of a varying character string that contains a list of option specifications separated by commas.

Processing notes

Option values are coded as *value* or *(value1,value2)*. A list of values is enclosed in parentheses. A value containing special characters is enclosed in single quotation marks. An imbedded single quotation mark is coded as two consecutive single

quotation marks. Special characters include all EBCDIC characters other than upper and lower case alphabetics, numerics, national characters (@ # $), and the underscore. YES and NO values can be abbreviated Y and N, respectively.

Options specified for a workmod override any corresponding options specified for that dialog. Options specified at the dialog level override the corresponding system defaults, and apply to all workmods within the dialog unless overridden. If INTENT=ACCESS, these keywords are not allowed: ALIGN2, CALL, CALLIB, EDIT, LET, MAP, OVLY, RES, TEST, XCAL, and XREF.

The options list specified in the PARMS= parameter is a character string identical to the PARM= value defined in the "Binder options reference" chapter of *z/OS MVS Program Management: User's Guide and Reference*, with the following restrictions:

- The list is not enclosed with apostrophes or parentheses
- Environmental options cannot be specified on SETO. See the list of environmental options in “Setting options with the binder API” on page 173
- The EXITS and OPTIONS options are also not allowed in this list.

The OPTION and OPTVAL operands are used together to specify a single option and its value.

- None of the environmental options can be specified. See “Setting options with the binder API” on page 173.
- The following invocation options may not be specified on the OPTION/OPTVAL operands of SETO because they are really mapped to something different: EXITS, OPTIONS, REFR, RENT, and the YES, NO, or default values for REUS.
- The negative option format (for example, NORENT) is not allowed. The corresponding option with a value must be used (for example, OPTION=REUS,OPTVAL=SERIAL).
- An option specified using the OPTION and OPTVAL operands overrides any value for that same option specified within the PARMS operand.

You can specify a z/OS UNIX System Services file as the CALLIB parameter value on a SETO call.

Return and reason codes

The common binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion.
08	83000109	One or more options designated as environmental have been specified on SETO. Option ignored.
12	83000100	Neither dialog token nor workmod token were specified. Request rejected.
12	83000106	The option specified is invalid for a workmod specified with INTENT=ACCESS. Request rejected.
12	83000107	Invalid option keyword specified. Request rejected.
12	83000108	The option value is invalid for the specified keyword. Request rejected.
12	83000113	An option you specified is valid only for the STARTD function. The request is rejected.

Parameter list

If your program does not use the IEWBIND macro, place the address of the SETO parameter list in general purpose register 1.

Table 27. SETO parameter list

```
PARMLIST    DS  0F
            DC  A(SETO)              Function code
            DC  A(RETCODE)           Return code
            DC  A(RSNCODE)           Reason code
            DC  A(DIALOG)            Dialog token
            DC  A(WORKMOD)           Workmod token
            DC  A(OPTION)            Option keyword
            DC  A(OPTVAL)            Option value
            DC  A(PARMS)             Options list
SETO        DC  H'20'                SETO function code
            DC  H'version'           Interface version number
```

Note: The PARMS parameter is an addition for Version 2 and X'80000000' must be added to either the OPTION parameter (for Version 1) or the PARMS parameter.

STARTD: Start dialog

STARTD begins a dialog with the binder, establishing the processing environment and initializing the necessary control blocks. You specify the ddnames for the data sets to be accessed, how errors are to be handled, and the global binder options.

STARTD returns a dialog token that is included with later calls for the same dialog.

The syntax of the STARTD call is:

```
[symbol]    IEWBIND    FUNC=STARTD

                       [,VERSION=version]
                       [,RETCODE=retcode]
                       [,RSNCODE=rsncode]
                       ,DIALOG=dialog
                       [,FILES=filelist]
                       [,EXITS=exitlist]
                       [,OPTIONS=optionlist]
                       [,PARMS=parms]
                       [,ENVARS=envars]
```

FUNC=STARTD
Specifies that a dialog is opened and initialized.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

Notes:

1. If VERSION=1 is specified for the STARTD call, PARMS cannot be specified as a macro keyword. The parameter list ends with the OPTLIST parameter (with the high-order bit set). This exception is for Version 1 only.
2. ENVARS cannot be specified if VERSION is less then 6.

RETCODE=*retcode* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **– RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that is to receive the reason code returned by the binder.

DIALOG=*dialog* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte area where the binder places the dialog token. This token must not be modified.

FILES=*filelist* **– RX-type address or register (2-12)**
Specifies the address of a list containing one entry for each binder file for which a ddname or file name is provided. You code some or all of these file names in the list and provide a ddname or filename for each:

File name	Description
CALLIB	Automatic call library
MODLIB	Target program library
PRINT	Listing data set for messages produced by the LIST, MAP, and XREF options
TERM	Terminal data set for messages issued during binder processing
SIDEFILE	Data set to contain the side file of a DLL module.

The *ddnames* specified for PRINT and TERM can designate z/OS UNIX System Services files. CALLIB can designate a z/OS UNIX System Services directory and a z/OS UNIX System Services archive file. MODLIB can designate a z/OS UNIX System Services directory. SIDEFILE can designate a z/OS UNIX System Services directory or a z/OS UNIX System Services file.

Entries for all files on this list will accept a z/OS UNIX pathname in place of a ddname. Path names must begin with a slash (**/**) or a period and a slash (**./**).

EXITS=*exitlist* **– RX-type address or register (2-12)**
Specifies the address of a list of user exit names. You can specify the following user exits in this list:

MESSAGE exit
Specifies an entry into the calling program that receives control immediately prior to the binder issuing a message. See “Message exit” on page 178.

SAVE exit
Specifies an entry into the calling program that receives control if the binder is about to reject a primary name or an alias name or after an attempt to save a member or alias name. The save operation might have succeeded or failed. See “Save exit” on page 178.

Note: This exit is not invoked if the target is a z/OS UNIX System Services file.

INTFVAL exit
The Interface Validation Exit (INTFVAL) allows your exit routine to examine descriptive data for both caller and called at each external reference. The exit can perform audits such as examining parameter

passing conventions, the number of parameters, data types, and environments. It can accept the interface, rename the reference, or leave the interface unresolved. See “Interface validation exit” on page 180.

See Chapter 8, “User exits,” on page 177 for additional information on writing user exit routines.

OPTIONS=*optionlist* **— RX-type address or register (2-12)**
Specifies the location of an options list that contains the address of binder options to be initialized during the STARTD call. Any option that can be set by SETO can be initialized by STARTD. See Chapter 7, “Setting options with the regular binder API,” on page 173 for a list of allowable options. However, the EXITS option listed in the table may not be specified as part of the OPTIONS parameter; use the EXITS or PARMS parameter on STARTD to specify user exits.

Note: The negative option format (for example, NORENT) is not allowed. Use the corresponding keyword with a value. For example:

```
DC   CL8'REUS'
DC   AL4 (6) ,C'SERIAL'
```

PARMS=*parms* **— RX-type address or register (2-12)**
Specifies the location of a varying character string that contains a list of option specifications separated by commas. The STARTD FILES, EXITS and OPTIONS lists have precedence over the STARTD PARMS string in the STARTD call. See “Setting options with the binder API” on page 173 for more information.

ENVARS=*envars* **— RX-type address or register**
Specifies the address of a list of 31-bit pointers. Each pointer in the list contains the address of a character string that is an environment variable, in the form of *name=value*, to be passed to the specified program. Each character string must end with zeros. Note that this is the same as passing the external variable, *environ*. See “Environment variables” on page 97 for more information.

Passing lists to the binder

Any list passed to the binder must conform to a standard format, consisting of a fullword count of the number of entries followed by the entries. Each list entry consists of an 8-byte name, a fullword containing the length of the value string, and a 31-bit pointer to the value string. The list specification is provided in Table 28.

Table 28. Binder list structure

Field Name	Field Type	Offset	Length	Description
LIST_COUNT	Integer	0	4	Number of 16-byte entries in the list
LIST_ENTRY	Structure	4,20,...	16	Defines one list entry
ENTRY_NAME	Character	0	8	File, exit, or option name
ENTRY_LENGTH	Integer	8	4	Length of the value string
ENTRY_ADDRESS	Pointer	12	4	Address of the value string

Note: ENTRY_NAME, ENTRY_LENGTH, and ENTRY_ADDRESS are repeated for each entry in the list up to the number specified in LIST_COUNT.

You code the data pointed to by ENTRY_ADDRESS according to the list type:

File list: Code one entry for each file name:

ENTRY_NAME: `'CALLIB  '`, `'MODLIB  '`, and so on. See list of file names in FILES parameter description in “STARTD: Start dialog” on page 91.

ENTRY_LENGTH: The byte length of the corresponding ddname.

ENTRY_ADDRESS: The address of the string containing the ddname.

Each file name specified in the FILES parameter of the STARTDialog API must correspond to a currently defined ddname. Your data sets can be new or preallocated. Although you can use any valid ddname for a given FILE name, the following ddnames are recommended. Their allocation requirements are listed below:

FILE name	Recommended ddname
CALLIB	SYSLIB
MODLIB	SYSLMOD
PRINT	SYSPRINT
TERM	SYSTERM
SIDEFILE	SYSDEFSD

The SYSLIB Data Set (the CALLIB file): This DD statement describes the automatic call library, which must reside on a direct access storage device. This is a required data set if you want to enable autocall processing while you bind your modules through the use of the BINDWorkmod API call (See the CALLIB parameter in the BINDW API).

The data set must be a library and the DD statement must not specify a member name. You can concatenate any combination of object module libraries and program libraries for the call library. If object module libraries are used, the call library can also contain any control statements other than INCLUDE, LIBRARY, and NAME. If this DD statement specifies a PATH parameter, it must specify a directory.

Table 29 shows the the SYSLIB data set attributes, which vary depending on the input data type.

Table 29. SYSLIB data set DCB parameters

LRECL	BLKSIZE	RECFM
80	80	F, FS, OBJ, XOBJ, control statements, and GOFF
80	32720 (maximum size)	FB, FBS OBJ, XOBJ, control statements, and GOFF
84+	32720 (maximum size)	V, VB, GOFF object modules
n/a	32720 (maximum size)	U, load modules
n/a	4096	U, program objects

The SYSLMOD data set (the MODLIB file): It is the target library for your SAVEWorkmod API calls when ACCESS=BIND on your CREATEWorkmod API call. That is, SYSLMOD is the library that contains your bound modules. As such, it must be a partitioned data set, a PDSE, or a z/OS UNIX System Services file.

Although a member name can be specified on the SYSLMOD DD statement, it is used only if a name is not specified on the SAVEWorkmod SNAME parameter. (See “SAVEW: Save workmod” on page 81.) Therefore, a member name should not be specified if you expect to save more that one member in a binder dialog. For additional information on allocation requirements for SYSLMOD, see SYSLMOD DD statement in *z/OS MVS Program Management: User's Guide and Reference*.

The SYSPRINT data set (the print file): The binder prints diagnostic messages to this data set. The binder uses a logical record length of 121 and a record format of FBA and allows the system to determine an appropriate block size.

Table 30 shows the data set requirements for SYSPRINT.

Table 30. SYSPRINT DCB parameters

LRECL	BLKSIZE	RECFM
121	121	FA
121	32670 (maximum size)	FBA
125		VA or VBA

The SYSTERM data set (the TERM file): SYSTERM defines a data set for error and warning messages that supplements the SYSPRINT data set. It is always optional. SYSTERM output consists of messages that are written to both the SYSTERM and SYSPRINT data sets, and it is used mainly for diagnostic purposes.

Table 31 shows the data set requirements for SYSTERM.

Table 31. SYSTERM DCB parameters

LRECL	BLKSIZE	RECFM
80	32720 (maximum size)	FB

The SYSDEFSD data set (the SIDEFILE file): When a module (call it module A) is enabled for dynamic linking through the DYNAM(DLL) binder option, a complementary file can be generated to go along with it. Module A becomes a DLL, and the complementary file becomes its *side file*. The side file is saved in the data set represented by the SYSDEFSD ddname. The side file contains the external symbols of DLL A, known as *exports*. These external symbols can be referenced by other DLLs and are known as *imports* to these modules. If module A does not export any symbols, no side file is generated for it. This applies to any DLL.

SYSDEFSD can be a sequential data set, a PDS, a PDSE, or a z/OS UNIX System Services file. If it is a sequential data set, the generated side files for multiple DLLs are appended one after another, provided that the DISP=MOD parameter is supplied in the SYSDEFSD ddname specification. If SYSDEFSD is a PDS or a PDSE, the side file is saved as a member with the same name as the DLL to which it belongs. Refer to the processing notes of the SAVEW API for additional information.

The SYSDEFSD DD statement is optional. However, when the ddname is absent, the binder issues a warning message if at bind time a program generates export records and the DYNAM(DLL) binder option has been specified.

Exit list:

ENTRY_NAME: `'MESSAGE '`, `'INTFVAL '`, or `'SAVE    '`.

ENTRY_LENGTH: 12.

ENTRY_ADDRESS: The location of a three-word area containing three addresses. See the message list addresses in “Message exit” on page 178, the save list addresses in “Save exit” on page 178 and the interface validation list addresses in “Interface validation exit” on page 180 for the contents of the three addresses.

Option list:

ENTRY_NAME: The option keyword (for example, `'LIST    '`, `'MAP     '`, `'CALLERID'`). Option keywords cannot be truncated and negative options cannot be specified (NOLIST, NOPRINT, and so on). If INTENT=ACCESS, these keywords are not allowed: ALIGN2, CALL, CALLIB, EDIT, LET, MAP, OVLY, RES, TEST, XCAL, and XREF.

ENTRY_LENGTH: The length of the option value as a character string; it can be zero.

ENTRY_ADDRESS: The address of the character string encoded with the option's value. The address can be zero. The maximum length of the option value string is 256 bytes. Use commas and parentheses if a sublist is required.

Return and reason codes

The common binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion.
04	83000112	The binder encountered an unsupported option in the OPTIONS file. The option is ignored.
04	83000204	The binder was unable to open the trace data set during initialization. Processing continues without trace.
08	83000108	An option value is missing or contains an invalid setting.
08	83000111	An OPTIONS option was encountered in the options file. The option is ignored.
08	83000200	The binder was unable to open the PRINT data set during initialization. Processing continues without PRINT.
08	83000201	One or more invalid options were passed on STARTD. Those options were not set, but processing continues.
12	83000203	The binder was unable to open the TERM data set during initialization. Processing stops.
08	83000205	The current time was not available from the operating system. Time and date information in printed listings and IDR records will be incorrect.
08	83000206	The binder was unable to open the SYSTERM data set because its DDNAME was not specified in the FILES parameter of STARTD. Processing continues without SYSTERM.

Return Code	Reason Code	Explanation
12	83000207	The binder was unable to open the SYSPRINT data set because its DDNAME was not specified in the FILES parameter of STARTD. Processing continues without SYSPRINT.

Environment variables

When the binder API is used by a program running in the z/OS UNIX subsystem, the environ parameter may be used to pass the C/C++ runtime variable of that name to the binder, in order to give the binder access to the array of environment variables. If a user sets binder environment variables (those documented below) in the UNIX shell, this is the only way the binder can get access to them. For further information about the C runtime 'environ' variable, refer to *z/OS XL C/C++ Run-Time Library Reference*, SA22-7821.

Although it is not recommended, you may also construct your own 'envars' parameter using the same format as that of the C runtime variable. In this case, 'envars' must be the address of an array of pointers. Each pointer is the address of a null-terminated string representing an individual environment variable in the form 'keyword=*value*'. The last array entry must be a null pointer.

The following UNIX shell environment variables are recognized by the binder. They are specified in the form:

```
export  NAME=value
```

where NAME is the name of the environment variable.

IEWBIND_PRINT
: the pathname or ddname to be used for SYSPRINT.

IEWBIND_TERM
: the pathname or ddname to be used for SYSTERM.

IEWBIND_OPTIONS
: the binder option string. These will be appended to options passed explicitly through the STARTD API call (and will take precedence). There are additional environment variables defined for the binder diagnostic data sets. See the "Binder Serviceability Aids" chapter in*z/OS MVS Program Management: User's Guide and Reference* for more information on passing them on STARTD.

Parameter list

If your program does not use the IEWBIND macro, place the address of the STARTD parameter list in general purpose register 1.

Table 32. STARTD parameter list

```
PARMLIST    DS      0F
            DC      A(STARTD)           Function code
            DC      A(RETCODE)          Return code
            DC      A(RSNCODE)          Reason code
            DC      A(DIALOG)           Dialog token
            DC      A(FILELIST)         File list
            DC      A(EXITLIST)         Exit list
            DC      A(OPTLIST)          Option list
            DC      A(PARMSTR)          Parameters
            DC      A(ENVARS)           Environment Variables
```

Table 32. STARTD parameter list (continued)

STARTD	DC	H'01'	STARTD function code
	DC	H'*version*'	Interface version number

Note: X'80000000' must be added to the last parameter. For version 1, that will be OPTLIST. For other versions it may be either PARMSTR or (beginning with version 6) ENVARS.

STARTS: Start segment

STARTS designates the beginning of an overlay segment when creating an overlay format program module. The OVLY option must have been specified for this workmod. OVLY-format modules can be saved only as load modules or PM1 format program objects.

The syntax of the STARTS call is:

```
[symbol]      IEWBIND     FUNC=STARTS

                          [,VERSION=version]
                          [,RETCODE=retcode]
                          [,RSNCODE=rsncode]
                          ,WORKMOD=workmod
                          ,ORIGIN=origin
                          [,REGION={YES | NO}]
```

FUNC=STARTS
Specifies the beginning of an overlay segment within an overlay format program.

VERSION=*1* | *2* | *3* | *4* | *5* | *6* | *7*
Specifies the version of the parameter list to be used. The default value is VERSION=1.

RETCODE=*retcode* **– RX-type address or register (2-12)**
Specifies the location of a fullword integer that is to receive the return code returned by the binder.

RSNCODE=*rsncode* **– RX-type address or register (2-12)**
Specifies the location of a 4-byte hexadecimal string that is to receive the reason code returned by the binder.

WORKMOD=*workmod* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte area that contains the workmod token for this request.

ORIGIN=*origin* **– RX-type address or register (2-12)**
Specifies the location of an 8-byte varying character string that contains the symbol that designates the segment origin. This symbol is independent of other external symbols in the workmod and has no relation to external names in the ESD.

REGION={YES | NO}
Specifies whether or not the segment begins a new region within the program module. This is an optional keyword. The argument can be abbreviated as **Y** or **N**. NO is the default.

Processing notes

A STARTS request is valid only when the processing intent is BIND.

For more information on overlay format programs, see the information on overlay programs in *z/OS MVS Program Management: User's Guide and Reference*.

Return and reason codes

The common binder API reason codes are shown in Table 3 on page 22.

Return Code	Reason Code	Explanation
00	00000000	Normal completion.
12	83000104	The STARTS function is not valid against a workmod specified for INTENT=ACCESS. Request rejected.
12	83000712	The maximum of 4 regions will be exceeded. Request rejected.
12	83000713	The maximum of 255 segments will be exceeded. Request rejected.

Parameter list

If your program does not use the IEWBIND macro, place the address of the STARTS parameter list in general purpose register 1.

Table 33. STARTS parameter list

```
PARMLIST    DS      0F
            DC      A(STARTS)               Function code
            DC      A(RETCODE)              Return code
            DC      A(RSNCODE)              Reason code
            DC      A(WORKMOD)              Workmod token
            DC      A(ORIGIN)               Segment origin symbol
            DC      A(REGION+X'80000000')   Region option and end-of-list indicator
STARTS      DC      H'35'                   STARTS function code
            DC      H'version'              Interface version number
REGION      DC      CL1                     Region option

                                                'Y'=Yes,  'N'=No
```

Binder API reason codes in numeric sequence

Table 34 shows all reason codes returned by the binder API. The API function(s) that return the codes are shown. Note that many of the codes can be the result of an INCLUDE call that includes additional control statements into the binder.

For a more detailed explanation of the specific error condition, refer to the Return and Reason Codes section of the particular API function being processed in Chapter 1, “Using the binder application programming interfaces (APIs),” on page 1.

Table 34. All binder API reason codes

Reason Code	API Function	Explanation
00000000	All API functions	Successful completion.
83000001	All API functions	Invalid workmod token. Request rejected.
83000002	All API functions	Invalid dialog token. Request rejected.
83000003	All API functions	Binder invoked from within user exit. Request rejected.
83000004	All API functions	Invalid function code specified. Request rejected.
83000005	All API functions	Invalid call parameter. Request rejected.

Table 34. All binder API reason codes (continued)

Reason Code	API Function	Explanation
83000006	All API functions	Requested function not allowed while in PUTD input mode. Request rejected.
83000007	All API functions	A symbol passed in the parameter list contains invalid characters.
83000008	All API functions	Wrong number of arguments specified. Request rejected.
83000009	All API functions	One or more parameters not accessible by the binder. Request rejected.
83000010	All API functions	Parameter list not addressable by the binder. Request rejected.
83000050	All API functions	WKSPACE storage limit exceeded. Dialog terminated.
83000051	All API functions	Insufficient storage available. Dialog terminated.
83000060	All API functions	Operating system at wrong level. Request rejected.
83000100	SETO	Neither Dialog nor Workmod Token were specified on a call. Request rejected.
83000101	GETE, INCLUDE, LOADW, PUTD, SETL	Invalid combination of parameters specified. Request rejected.
83000102	GETD, GETE, GETN	Workmod was in an unbound state. GET calls not allowed. Request rejected.
83000103	INCLUDE	INTENT = BIND and INTYPE not equal to NAME. INCLUDE request rejected.
83000104	ALTERW, BINDW, INSERTS, ORDERS, SETL, ALIGN, STARTS	Function invalid for INTENT = ACCESS. Request rejected.
83000106	SETO	Option invalid for INTENT = ACCESS. SETO request rejected.
83000107	SETO	Invalid Option keyword specified. SETO request rejected.
83000108	SETO, STARTD	The option value is invalid for the specified keyword. Request rejected.
83000109	SETO	The OPTION cannot be specified on a SETO function call. Request rejected.
83000111	STARTD	Syntax error or unrecognized option in parameter list.
83000112	STARTD	Obsolete option specified in parameter list and is ignored.
83000113	SETO	OPTION can be specified only when using the batch entry points.
83000200	STARTD	Unable to open print data set during initialization. Processing continues without print.
83000201	STARTD	One or more invalid options, specified on STARTD, were ignored. Processing continues.
83000203	STARTD	Unable to open SYSTERM data set during initialization. Processing continues without SYSTERM.

Table 34. All binder API reason codes (continued)

Reason Code	API Function	Explanation
83000204	STARTD	Unable to open IEWTRACE data set during initialization. Processing continues without IEWTRACE.
83000205	STARTD	Unable to obtain date and time from operating system. Date and time stamps set to zero or blanks on modules and listings.
83000206	STARTD	Data set or file allocated to TERM could not be found. Processing continues without PRINT.
83000207	STARTD	Data set or file allocated to PRINT could not be found. Processing continues without PRINT.
83000300	BINDW	Unresolved references exist in the module. Either NCAL, NOCALL, or NEVERCALL were specified. Workmod has been bound.
83000301	BINDW	Unresolved references exist in the module. The names could not be found in the designated library. Workmod has been bound.
83000302	BINDW	Unresolved references exist in the module. No call library specified. Workmod has been bound.
83000303	BINDW	A reference module was located in the call library, but could not be included. Workmod has been bound with unresolved references.
83000304	BINDW	A name specified on an INSERT request was not resolved, or was resolved to a label that is not a section name. Insert request ignored.
83000305	BINDW	A name specified on an ORDER request was not resolved, or was resolved to a label that is not a section name. Order request ignored.
83000306	LOADW, SAVEW	The Save or Load Operation Summary could not be printed. Module might or might not have been processed correctly, depending on other factors.
83000307	BINDW	The Map or Cross Reference listings could not be printed, but the module was bound successfully.
83000308	BINDW	Unresolved references exist in the module. A member of the same name was located in the library, but did not resolve the symbol. Module was bound successfully.
83000309	BINDW	A name specified on an ALIGNT request was not resolved, or was resolved to a label that is not a section name. ALIGNT request ignored.
83000310	BINDW	One or more ALTER requests were pending at entry to autocall, and were ignored.
83000311	BINDW	Overlay option not specified, but module contained multiple segments. Overlay structure ignored.
83000312	BINDW	Root segment contains no text.
83000313	BINDW	A 3-byte VCON in an overlay module cannot be relocated correctly.
83000314	BINDW	Overlay module bound with at least one valid exclusive call.

Table 34. All binder API reason codes (continued)

Reason Code	API Function	Explanation
83000315	BINDW	Invalid exclusive calls in module. Relocation not possible. Module bound.
83000316	BINDW	Overlay specified, but module contained only one segment. Module bound, but not in overlay format.
83000317	BINDW	One or more valid exclusive calls present, but XCAL was not specified. Module bound in overlay format.
83000318	BINDW	Module contains no branches out of the root segment. Module bound in overlay format.
83000319	BINDW	Interface mismatch between caller and callee. The mismatch might be the XPLINK attribute, AMODE 64 and non-AMODE 64, code and data, or a signature field mismatch.
83000320	BINDW	An autocall library was not usable. Autocall proceeded without library.
83000321	BINDW	RMODE(SPLIT) or OVERLAY incompatible with COMPAT specification.
83000322	BINDW	Conflicting attributes specified for data elements belonging to a class. Data for the class was discarded.
83000400	SAVEW	Module has been saved, but has been marked NOT-EDITABLE.
83000401	SAVEW	One or more aliases could not be added to directory. Module saved successfully.
83000402	SAVEW	Specified entry name not defined in the module, or entry point offset is beyond the bounds of the containing element. Entry point defaults to first text byte.
83000403	SAVEW	Reusability of one or more sections was less than that specified for the module.
83000404	SAVEW	The module exceeded the limitations for included load modules. Module not saved.
83000405	SAVEW	A permanent write error was encountered while attempting to write the module. Module not saved.
83000406	SAVEW	A permanent read error was encountered while processing an input data set.
83000407	SAVEW	No valid member name was available during SAVEW call. Module not saved.
83000408	SAVEW	Workmod marked not-executable and cannot replace executable version. Module not saved.
83000409	SAVEW	A member of the same name exists in the library, but replace was not specified. Module not saved.
83000410	SAVEW	Error encountered converting module to requested output format.
83000411	SAVEW	SCTR specified for a program object. Option ignored.
83000412	SAVEW	A valid entry point could not be determined.
83000413	SAVEW	One or more external references were bound to modules in LPA. Module not saved.

Table 34. All binder API reason codes (continued)

Reason Code	API Function	Explanation
83000414	SAVEW	The workmod is null. No modules were successfully included from any source file. The workmod cannot be saved.
83000415	BINDW, LOADW, SAVEW	The module contains no text or no ESDs.
83000416	SAVEW	No DDNAME has been provided for the output library. Module not saved.
83000417	SAVEW	Target data set of SAVEW is not a library. Module not saved.
83000418	SAVEW	Target data set of SAVEW does not have a valid record format for a load library. Module not saved. **Note:** This can happen if the output PDS or PDSE has a record format specified other than U.
83000419		Incompatible options existed at save time for OVERLAY.
83000420	SAVEW	Workmod modified before being stored in load module format.
83000421	SAVEW	Module has more than 1 GIGABYTE of text. Module not saved.
83000422	SAVEW	Workmod has data that cannot be saved in the requested format.
83000423	SAVEW	DYNAM(DLL) was specified and exported symbols exist but SYSDEFSD was not allocated.
83000424	SAVEW	I/O error on SYSDEFSD.
83000425	SAVEW	Can't write to SYSDEFSD because the member name is too long.
83000426	LOADW, SAVEW	LISTPRIV option specified and unnamed sections exist.
83000427	SAVEW	An attempt to change the file attributes or a user ID or group ID failed while saving to a z/OS UNIX file.
83000500	INCLUDE	An INCLUDE call specifies a nonempty target workmod with INTENT=ACCESS. Request is rejected.
83000501	BINDW	Control statements included during autocall have been ignored.
83000502	INCLUDE	Some of the editing requests(CHANGE, DELETE, REPLACE) failed. Module included, but some changes not made.
83000503	INCLUDE	I/O error encountered while attempting to read data set or directory. Input not usable.
83000504	INCLUDE	Aliases and/or attributes were not included because directory not accessible; however, module included successfully.
83000505	INCLUDE	Included module marked NOT-EDITABLE and has been bypassed.
83000506	INCLUDE	Attempt to include object module into workmod with INTENT=ACCESS. Request rejected.

Table 34. All binder API reason codes (continued)

Reason Code	API Function	Explanation
83000507	INCLUDE	A format error has been encountered in an included module. The module has been bypassed.
83000509	INCLUDE	An attempt has been made to include a file of control statements when INTENT=ACCESS. Request rejected.
83000510	INCLUDE	Errors were detected in the included module. Module was bypassed.
83000511	INCLUDE	A control statement attempted to include a file that was already in the include path. The recursive include has been bypassed.
83000512	INCLUDE	More than one module in an included file, and INTENT=ACCESS was specified. Request rejected.
83000513	INCLUDE	I/O error encountered reading a file or directory. File not included.
83000514	INCLUDE	Data set or member not found. Module not included.
83000515	INCLUDE	Unsupported control statement in included file. Statement ignored.
83000516	INCLUDE	Format error encountered in included control statement. Error statements ignored.
83000517	INCLUDE	Name statement encountered, but output library (SYSLMOD) not specified. Statement rejected.
83000518	INCLUDE	Name statement encountered in secondary input file. Statement ignored.
83000519	INCLUDE	Included module contained errors and has been bypassed.
83000520	INCLUDE	Data set or member specified on control statement could not be found. Include has been bypassed.
83000521	INCLUDE	An I/O error has been encountered while attempting to read the data set or directory. Include bypassed.
83000522	INCLUDE	The data set specified on an include statement could not be opened. Include bypassed.
83000523	INCLUDE	For INTENT=ACCESS, the requested module contained a format error and was not placed in the workmod.
83000525	INCLUDE	An unusual condition was encountered while processing and EDIT request (CHANGE, DELETE, REPLACE).
83000526	INCLUDE	An unusual condition was encountered while processing an input module.
83000527	INCLUDE	Target section on IDENTIFY control statement does not exist.
83000528	INCLUDE	The DE parameter value; the TTR or K (concatenation) field is invalid.
83000550	ALTERW	A section for which an EXPAND request was made was not in the workmod. Request not processed.
83000551	ALTERW	The name on an EXPAND request matches a symbol that is not a section in the workmod. Request not processed.

Table 34. All binder API reason codes (continued)

Reason Code	API Function	Explanation
83000552	ALTERW	The name on a CHANGE or REPLACE request is blank. Request rejected.
83000553	ALTERW	An EXPAND request was made for more than 1 GIGABYTE.
83000554	ALTERW	The class requested for expanding does not exist. Request rejected.
83000555	ALTERW	The class requested for expanding is not text. Request rejected.
83000556	EXPAND	The name on an EXPAND request did not match that of a section in workmod. Request not processed.
83000600	SAVEW	Output library (SYSLMOD) not found. Request rejected.
83000601	SAVEW	Unable to close output library. Module saved but might be unusable.
83000602	SAVEW	Unable to open output library. Save failed.
83000603	LOADW, SAVEW	The AMODE or RMODE of one or more sections is not compatible with the AMODE or RMODE of the primary entry point. Save or load operation continues.
83000604	LOADW, SAVEW	AMODE/RMODE combination for the module is invalid. Save or load operation continues.
83000605	LOADW, SAVEW	No entry point available from input. Defaults to first text byte.
83000606	LOADW, SAVEW	One or more RMODE(24) sections have been included in an RMODE(ANY) module.
83000607	LOADW, SAVEW	The module was loaded successfully, but the indicated 2-byte adcon(s) did not relocate correctly.
83000650	LOADW	Entry name not defined in module. Defaults to first text byte.
83000651	LOADW	IDENTIFY failed because name already known to system. Load successful, but module cannot be reached through system linkage.
83000652	LOADW	Insufficient virtual storage to load the module. Module not loaded.
83000653	LOADW	An error of severity greater than the LET option was seen while processing load.
83000654	LOADW	An error was found while converting from workmod to executable form.
83000655	LOADW	Extent list buffer provided room for only one extent, but a second extent exists for the mini-cesd. Module loaded successfully.
83000656	LOADW	The module was bound in overlay format and cannot be loaded.
83000657	LOADW	An AMODE(24) module has been bound to one or more modules in ELPA.

Table 34. All binder API reason codes (continued)

Reason Code	API Function	Explanation
83000702	ALTERW	The name specified on an IMMEDIATE ALTERW request could not be found in the ESD. Alteration rejected.
83000704	ENDD	Unexpected condition during ENDD. Dialog ended, but some resources might not have been released.
83000705	GETE	Specified symbol could not be located in workmod. No data returned in buffer.
83000706	ALTERW	The new name specified on an IMMEDIATE CHANGE request already existed. The name or section was deleted, and the requested change made.
83000707	DELETEW	The workmod was in an altered state, but PROTECT=YES was specified. DELETEW request rejected.
83000708	ENDD	One or more workmods were in an active state, but PROTECT=YES was specified. ENDD request rejected.
83000709	RESETW	The workmod was in an altered state, but PROTECT=YES was specified. RESETW request rejected.
83000710	ALIGNT	Name already specified on ALIGNT request. Request rejected.
83000711	ADDA, INSERTS, ORDERS, SETL	A previous INSERT, ORDERS, ADDA or SETL request for this name has been replaced.
83000712	STARTS	Maximum of 4 regions will be exceeded. Request rejected.
83000713	STARTS	Maximum of 255 segments will be exceeded. Request rejected.
83000719	BINDW	Module contains no text.
83000750	GETD, GETN	Buffer too small for one record.
83000800	GETD, GETE, GETN	End of data condition. Some data returned in buffer.
83000801	GETD, GETE, GETN, PUTD	Requested element not found or empty. No data returned.
83000802	PUTD	Attempt to create new section with INTENT=ACCESS. Request rejected.
83000803	PUTD	Target section for PUTD not found. Request rejected.
83000804	PUTD	Attempt to modify ESD or RLD with INTENT=ACCESS. Request rejected.
83000805	PUTD	Attempt to extend length of text with INTENT=ACCESS. Request rejected.
83000806	PUTD	Attempt to modify binder generated sections. Request rejected.
83000807	PUTD	Attempt to modify existing section when NEWSECT=YES, or NEWSECT=NO was specified while still in input mode. Request rejected.
83000808	PUTD	Attempt to modify IDRB record. Request rejected.

Table 34. All binder API reason codes (continued)

Reason Code	API Function	Explanation
83000810	GETN	CURSOR negative or beyond end of item.
83000811	PUTD	One or more errors detected in module just completed. Section(s) not merged.
83000812	GETE	OFFSET negative or beyond end of item or module. No data returned in buffer.
83000813	GETD	Workmod data is incompatible with the specified buffer version.
83000814	PUTD	Severe data errors in records contained in PUTD data buffers.
83000815	PUTD	Errors (such as invalid names) found in records contained in PUTD data buffers. Some data might have been dropped.
83000816	BINDW	Classes C_WSA and C_WSA64 are both present in the module. z/OS Language Environment does not support the presence of these two classes in the same program object, and the resulting module will not execute correctly.
83002342	GETC	Some of the passed compile unit numbers do not exist in workmod. Data for the valid compile units is returned.
83002349	GETD	Not all adcons are successfully relocated. This condition could occur because relocation addresses for all the segments are not passed, or because the adcon length is insufficient to contain the address.
83002375	GETD	The class is not a text class.
83002379	GETD	Binder encounters a bad cursor for class B_PARTINIT and processing has been stopped.
83002481	BINDW	The instruction address or the target address is not even.
83002492	BINDW	The operand of the instruction exceeds the destination range.
83002493	BINDW	A relative immediate instruction with multiple external symbols is encountered.
83002494	BINDW	Certificate setup problem.
83002495	BINDW	The binder could not resolve a weak reference used by a relative immediate address constant.
83002496	BINDW	A relative immediate address constant references an unresolved symbol. This is not supported.
83002497	BINDW	The binder expected a symbol to be resolved from a specific library member but it was not.
83002622	SAVEW	The signed module is saved to the format that does not support signing.
83002623	BINDW, SAVEW	Unexpected return code from the RACF call.
83002624	BINDW, DELETEW, RESETW	Unexpected error from SIGCLEAN.
83000FFF	ALL	IEWBIND module could not be loaded. Issued by IEWBIND invocation macro.

Table 34. All binder API reason codes (continued)

Reason Code	API Function	Explanation
83EE2900	ALL	Binder logic error. Dialog terminated.
83FFaaa0	ALL	Binder abend aaa occurred. Dialog terminated.

Chapter 4. IEWBFDAT - Binder Fast data access API functions

This topic describes the fast data access service that allows you to more efficiently obtain module data from a program object. The fast data access services differ from the binder API described in “IEWBIND function reference” on page 24 in a number of ways. The primary differences are:

Fast data access	Binder API
Used to inspect or obtain information about an existing program,	Also provides access to all other binder services, including creation of new programs and updates to existing ones.
Supports program objects stored in UNIX files or a PDSE except for PO1 overlay modules.	Also supports traditional load modules stored in a PDS, and all object file formats.
Is optimized for minimum overhead in accessing data as it exists on DASD.	Establishes a robust recovery environment on every call. Converts all data available to it into a standardized internal format.
Utilizes 64-bit storage to hold the program object, and because of that needs very little 31-bit or 24-bit storage.	Utilizes a data space to hold the program object as it exists on DASD, but also needs large additional amounts of 31-bit storage for the standardized internal format and other purposes.

Note: The lack of fast data access support for load modules in a PDS may appear to be a severe restriction, but that format has been stabilized and is completely documented in Appendix B, “Load module formats,” on page 195, so applications can process it themselves. Fast data access to PDS load modules would not be able to approach the performance of direct application access.

Using the fast data access service

Fast Data access services are usable from any language that supports the required data types. The services may be requested using either of two call interfaces: Request Code and Unitary. The Request Code interface was introduced in z/OS V1.R5. The Unitary interface is retained for compatibility with earlier releases and is functionally stabilized. IBM recommends that new applications use the Request Code interface.

Differences between the Request Code and Unitary interfaces include:

1. If only a limited amount of information is needed, the Unitary interface can provide it in a single request, while the Request Code interface always requires at least three requests.
2. The Unitary interface may simplify the calling code in some cases, but should not be thought of as more efficient. To implement it the fast data code analyzes the parameters and breaks the call into a sequence of Request Code calls to the actual processing routines.
3. The Unitary interface call requires 15 or 16 positional parameters, though for assembler programs the IEWBFDA macro can be used to generate the parameter list. The Request Code interface requires from two to eight positional parameters.

© Copyright IBM Corp. 1991, 2012

4. The IEWBFDA macro, which takes care of locating the fast data Code, can be used only by assembler programs and only with the Unitary interface. The special C/C++ support, which also takes care of locating the fast data Code, can be used only with the Request Code interface.
5. Compile unit information, section name lists, and class name lists can only be obtained using the Request Code interface.

C/C++ programs should use the special interface support provided for that language. See Chapter 5, “IEWBNDD, IEWBNDDX - Binder C/C++ API DLL functions,” on page 131 for more information. This provides a variant of the Request Code interface that is more natural in a C/C++ environment. It uses, for example, null-terminated strings rather than the length-prefixed strings defined here.

With either call interface, the code being called is entry point IEWBFDAT in a separate load module in the system Link Pack Area. The application making the calls must normally use some language-dependent means, such as the LOAD SVC, to locate IEWBFDAT, and use that address for the call. The special C/C++ support takes care of that requirement for the application. For assembler programs using the Unitary interface, an optional IEWBFDA macro is provided which LOADs and DELETEs IEWBFDAT as well as generating the parameter list and making the call.

The services described here return data in the same formats used by the binder API described in “IEWBIND function reference” on page 24, so the information in Chapter 2, “IEWBUFF - Binder API buffers interface assembler macro for generating and mapping data areas,” on page 13 about generating and mapping data areas applies equally to fast data access. The buffer formats defined in Appendix D, “Binder API buffer formats,” on page 239 also apply to these services. See the Buffer parameter description below for more details.

Environment

Your program's environment must have the following characteristics before invoking the service:

- Enabled for I/O and external interrupts
- Holds no locks
- In task control block (TCB) mode
- With PSW key equal to the job step TCB key
- In primary address space mode
- In 31-bit addressing mode
- In either supervisor or problem program state
- Authorization to get enough 64-bit storage available to hold the stored program object

All requests are synchronous. The service returns control to your program after the completion of the requested service. Services cannot be requested in cross-memory mode.

All addresses passed to the service in the form of parameters must be valid 31-bit addresses.

Parameter descriptions

There are variations in the interface to this service. The methods of passing parameters differ, but the same type of information is required in each case. This section discusses the formats and meaning of the data passed and returned, independent of the specific interface used.

Buffer

This refers to a structured area within which data is returned by the fast data access service. The calling application must provide specific control information in the buffer before passing it to a service. Chapter 2, “IEWBUFF - Binder API buffers interface assembler macro for generating and mapping data areas,” on page 13 defines an IEWBUFF assembler macro that can be used to allocate, map, and/or format the buffer correctly (see that topic for details). Appendix D, “Binder API buffer formats,” on page 239 contains details of all buffer type formats. Note that multiple buffer format versions exist. These should not be confused with program object versions. With a few exceptions, fast data is able to map data from any program object version into any buffer format version; however, earlier buffer format versions may be missing fields for features available in more recent program objects. In general, an application should use the most recent buffer version supported by the systems the application is designed to run on. Earlier PO formats can always be represented in more recent buffer formats.

There were major changes between PO1 and PO2 object formats, and corresponding major changes between version 1 and version 2 buffers. Because of this, version 1 ESD, RLD, and NAME buffers cannot be used to access more recent program object versions. If using IEWBUFF to generate buffers, please note that for compatibility the default remains VERSION=1. Applications should override that default in most cases.

Every buffer type has a header which identifies the type and version of the buffer. Three other fields within the buffer headers deserve special note:

BUFFER_LENG
: The total length of the buffer in bytes, including the header. Data entries are built following the header, but the total buffer size must also include space, in many buffer types, for name strings. These are built in descending address order working back from the end of the buffer. It should be noted that the binder supports external symbol names up to 32,767 bytes long, and that some entry types have multiple names for a single entry. An application written to support arbitrary names needs to provide a buffer which is large enough to handle any combination of names associated with a single entry. An ESD entry, for example, can have up to four names associated with it, so in theory the buffer might have to be more than 128 KB long to retrieve a single entry.

ENTRY_LEN
: Data will be returned as an array of entries. For example, External Symbol Dictionary entries are currently from 56 to 96 bytes long, depending on buffer version, but within a single buffer all ESD entries are of the same length. The length is an attribute of the buffer version, and must correspond with it. A few buffer types, including text buffers, effectively have unstructured or varying length blocks of data. These buffer types always use an entry length of 1, meaning that their content is counted in bytes.

ENTRY_COUNT
: This is a nominal number of entries that can be expected to fit in the buffer. The product of the entry count and the entry length is the size anticipated to be used by returned data, excluding symbol names. The entry count is not updated on return from a service, so it does not indicate the number of entries that are actually in the buffer. See the Count parameter below for that. The count returned could be less than this entry count even when there are more entries that have not yet been returned. This would happen if the name strings for the entries returned require more space than was reserved at the end of the buffer

for them. In some cases fast data may decide to go beyond the anticipated data area and return more entries than indicated by entry count, if additional space is available within the buffer.

Class

Class is passed as a character string with a leading two-byte binary length field. The length is the number of bytes in the class name, excluding the length field itself and any additional unused bytes in the field at the end of the name. See “Understanding binder programming concepts” on page 1 for a definition of classes in program objects, and a list of commonly used class names. B_PMAR is also accepted as a class name for the GD call, although it is not an actual class in a binder workmod.

Count

Count is a fullword output parameter specifying how many entries are returned in a buffer from the current call. Note that the length of one entry is specified within the buffer, as discussed above under Buffer.

Cursor

Cursor is a fullword which is both an input and an output field. Like Count, it is maintained in terms of entries, not bytes (except for buffer types whose entry length is 1). An application would normally set the cursor to 0 when beginning a sequence of accesses. At the end of each request the fast data access service will set the cursor to the next sequentially available entry. In most cases the application can reset the cursor to any desired value so long as it is within the scope of the objects being retrieved.

There are a few exceptions for entries which are structured but varying in length, such as PTI entries. These entries have a formal entry length of 1, but a practical entry length larger than that. If the application resets the cursor it must be set to the beginning of an entry. This means that for these classes the application could back up to a previously saved point, but it could not jump ahead. Any data skipped should be treated by the calling application as containing the fill character (normally X'00').

Notes on retrieving data from a PO1 module

The following are considerations to make when retrieving PO1 data:

- If ESDs from a PO1 are being retrieved using a version 2 or higher buffer, multiple ESDs records may be associated with a single cursor value. The cursor always indexes the records in the program object, while the count (as always) reflects the number of ESD records returned. In a very small buffer, it may not be possible to return all the records associated with one cursor value. In that case the reason code is 10800022, the count is zero, and the cursor is unchanged. An incomplete set of ESD records may be visible in the buffer.
- The class and section names retrieved using the GN fast data call for a PO1 format program object may have some minor differences from those reported by AMBLIST for the same module. This is because fast data retrieves data directly from the stored program object while AMBLIST retrieves data from a program object converted to a common format. In particular, binder-generated section x'00000003' (printed as $PRIV000003) does not exist in a PO1 module. The class which would normally be associated with this section, B_PRV, is also absent in PO1 format.

DCBptr

This is a pointer to a DFSMS Data Control Block. See DFSMS Macro Instructions for Data Sets.

Notes:

1. In most cases, this will be a DCB opened by the calling application. It is possible to use a DCB (such as a STEPLIB DCB or the LINKLIB DCB) opened by the MVS system, but there may be some additional restrictions when this is done. See “SB - Starting a session with a BLDL identifier” on page 117 for more information.
2. The fast data access service supports all data sets allocated in the extended addressing space (EAS) of an extended address volume (EAV).
3. The fast data access service supports the following dynamic allocation (DYNALLOC or SVC 99) options for all data sets: S99TIOEX(XTIOT), S99ACUCB (NOCAPTURE), and S99DSABA (DSAB above the line).

DDname

The name of a DD statement that identifies a PDSE data set, a concatenation of PDS and PDSE data sets, or a UNIX path. Note that fast data cannot process PDS members in a mixture of PDS and PDSE. Although a DD name is normally thought of as an 8-character field, it is treated here as a varying length character string beginning with a two-byte binary length field. The length is the number of bytes in the ddname, excluding the length field itself.

Notes:

1. The fast data access service supports all data sets allocated in the extended addressing space (EAS) of an extended address volume (EAV).
2. The fast data access service supports the following dynamic allocation (DYNALLOC or SVC 99) options for all data sets: S99TIOEX(XTIOT), S99ACUCB (NOCAPTURE), and S99DSABA (DSAB above the line).

DEptr

This refers to an individual member entry within the structure returned by BLDL, beginning with an eight-character member name. BLDL allows entries as short as 12 bytes, but entries passed to releases of fast data access services prior to z/OS V1R9 must be at least 62 bytes long. Beginning with z/OS V1R9, fast data uses only the first eight bytes of the entry, which contain the member name. If your code will never run on earlier z/OS releases you can bypass the BLDL call and pass a member name padded with blanks in an eight-character field.

Note: This upwardly compatible change applies only to the interfaces defined in this topic. The binder API defined in “IEWBIND function reference” on page 24 continues to require a 62-byte BLDL entry.

Eptoken

An eptoken is an 8-byte field returned by CSVQUERY. Use of CSVQUERY followed by a call to the fast data access service is often useful when dealing with programs that are already loaded, especially if their source is unknown. Given an eptoken, fast data can locate and usually access the true source of a loaded program even if it has been replaced on DASD since it was loaded. Beginning with z/OS V1R12, fast data access service supports eptokens associated with z/OS UNIX path names.

Member

This is normally the name of a program directory entry in a PDSE. Fast data also permits use of a member parameter with UNIX System Services files. In this context it is treated as an extension of the path name, whether provided in a Path parameter or located through a DD statement. When used this way the member string is simply appended to the path string, and may include subdirectory information. Member is passed as a varying length character string beginning with a two byte binary length field. The length is the number of bytes in the member name, excluding the length field itself.

Mtoken

The mtoken is common to all interfaces and calls. It is a fullword field provided by the application but with a value set by the fast data access service. An application can have multiple concurrent instances of interaction with fast data. Each instance deals with a single program being inspected, and each has a unique mtoken. The first time the application calls for a particular program it must set the mtoken to 0. The value returned must be used for all subsequent calls up through the call in which the application indicates it is terminating the instance.

Path

A UNIX file may be referred to either by passing a path name directly or by passing the name of a DD statement containing a PATH parameter. See also Member for a special fast data extension to path processing. Path is passed as a varying length character string beginning with a two byte binary length field. The length is the number of bytes in the path name, excluding the length field itself.

Retcode

Retcode is a fullword binary return code from the service. Fast data return codes are multiples of 4, from 0 to 16.

Rsncode

Rsncode is a four byte binary value providing more detail about the service return code. Fast data reason codes all begin with X'1080'. See “Return and reason codes” on page 128.

Section

This is the name of a control section within the program object being inspected. In most cases control section names are optional, but may be used to restrict the amount of data returned. There is an important relationship between Section and Cursor. When Section is omitted, Cursor is a position within the entire scope of the data. When Section is provided, Cursor is a position within the subset of the data defined by that section name. Section is passed as a varying length character string beginning with a two byte binary length field. The length is the number of bytes in the section name, excluding the length field itself.

The Request Code interface

The Request Code interface operates in a manner similar to the binder API in that a series of calls is required to extract data; at a minimum the application must make one call to start a session, one to get data, and one to end the session. The caller provides a parameter list for each call that specifies the service being requested. This is a standard call using OS linkages with two qualifications:

- The parameter lists must be constructed in the variable length form, with the high order bit set on the last parameter. In assembler this would be done using LINK or (preferably) CALL with the VL parameter. High level languages typically, though not universally, follow this convention by default.
- The program being called, IEWBFDAT, must not be resolved statically when the application is created, and is not a DLL. It is located in the MVS Link Pack Area and needs to be located dynamically during execution. In assembler language this is done using LOAD or LINK. High Level Language techniques for accomplishing this vary.

Every call provides at least three pieces of information:

- A function code. Two upper case EBCIC alphabetic characters.
- An interface level. Currently always 1.
- An mtoken. See “Mtoken” on page 114.

Upon return from fast data access, you can examine the return and reason codes. Fullword return and reason codes are returned in registers 15 and 0 respectively. For languages that do not provide access to one or both of those registers on return, an RC service call is provided, which returns the most recent return and reason codes in fields passed as parameters to the RC service call.

Function code usage summary

A fast data session begins with one of the “Start” service calls:

Function code	The program to be inspected is identified by
“SB - Starting a session with a BLDL identifier” on page 117	An open DCB and a BLDL directory entry
“SJ - Starting a session with a DD name or path” on page 118	Some combination of a ddname, a member name, and/or a path name
“SQ - Starting a session with a CSVQUERY token” on page 118	An entry point token returned by the MVS CSVQUERY service
“SS - Starting a session with a System DCB” on page 119	Equivalent to SB; retained for compatability

The session continues with some combination of data requests:

Function code	Type of data requested
“GC - Getting Compile unit information” on page 119	Compile unit information. This includes primarily data passed to the binder by the compiler on the source of each program making up the program object being inspected, but it also includes information on the DASD location of the program object itself.

Function code	Type of data requested
"GD - Getting Data from any class" on page 120	Generalized access to any data in any class in the program object. This includes loadable data such as the executable program and its Working Storage Area, but also non-loadable data such as relocation dictionaries or debugging classes inserted by the compiler.
"GE - Getting External Symbol Dictionary data" on page 121	A specialized form of GD for the External Symbol Dictionary class.
"GN - Getting Names of sections or classes" on page 122	Class or section names. Although this comes last alphabetically it is often the first data request issued, as it returns data that may be needed for issuing GD or GE service calls.

There are also two special purpose requests:

Function code	Purpose
"EN - Ending a session" on page 123	Terminate a fast data access session. This allows the service to release resources which it has been holding across multiple calls.
"RC - Return Code information" on page 123	Obtain return and reason codes for the most recent request. This service is used in languages without direct access to registers 15 and 0 after a service call.

Common Parameters

The following parameters common to all calls.

function code and interface level
A 4-byte character string consisting of 2 characters indicating the function being requested followed by a value indicating the interface level. The only valid value for interface level in this release is X'0001'.

mtoken
A 4-byte field that contains the module token provided by fast data. This token must be initialized to binary zero before the first call of a sequence.

Optional parameters

The common parameters listed in "Common Parameters" are always required. Most service calls have additional parameters, and most of them are required, but a few parameters are listed as optional. An optional parameter may be shown as omitted in one of three ways:

- If no parameters follow it, by passing a short parameter list. As noted earlier, all parameter lists are to be constructed in the VL or varying length form.
- Providing binary zero in place of an address in the parameter list.
- By passing a null value. The null value for a varying length character string such as SECTION, is a zero-length string. The null value for fullword or doubleword parameters is a fullword or doubleword of binary zero.

SB - Starting a session with a BLDL identifier

The calling application identifies the program object by providing an open DCB and an entry returned by the MVS BLDL service.

Table 35. SB parameter list

Parameter	Usage	Format	Content
1	in	structure	'SB', X'0001'
2	in/out	binary word	mtoken Must contain zero when the service is called. The service will supply a value if the service is successful.
3	in	structure	DCB **Note:** The parameter list points directly to the DCB, not to a pointer variable
4	in	structure	Member name or BLDL directory entry; only the first 8 bytes are inspected. **Note:** The parameter list points directly to the directory entry**Caution**: Releases of z/OS prior to V1R9 require a full BLDL entry here of at least 62 bytes, beginning with the member name.

This service is primarily designed for DCBs opened within the job step in the job step key. It can, however, also be used for DCBs opened by MVS, such as JOBLIB/STEPLIB, LINKLIB, or PGM=*.DD.

The service makes a number of tests to determine if the DCB passed by the caller can be used. These tests involve the access method and record format specified when the DCB is opened, the key in the DCB that is opened, and the current program authorization level. If the DCB cannot be used, the service attempts to open a DCB for the same DDname. If the DDname exists, but the OPEN fails, the request is rejected. Reason codes are expected from an SJ request rather than an SB request.

If the caller's DCB cannot be used, and the DDname does not exist for the current job step (as is the case for the LINKLIB DCB), a special authorized service is used to read the program object. In this case, the program object image is maintained in 31-bit storage within the address space rather than in 64-bit storage. This can change the storage requirements significantly.

Sample assembler code

```
         CALL  (15),(SBIL,MTOKEN,MYDCB,DE),VL

SBIL     DC    C'SB',X'0001'
MTOKEN   DC    F'0'          Replaced by the service
MYDCB    DCB   DDNAME=PDSE,DSORG=PO,RECFM=U,MACRF=R
BLDLIST  DC    H'1',H'62'    62 is the minimum for fast data
DE       DC    CL8'MEMBERNM'
         DS    CL50          Filled by BLDL
```

SJ - Starting a session with a DD name or path

The calling application identifies the program object either directly by a UNIX path name or indirectly by the name of a DD statement which identifies either a UNIX path or one or more partitioned data sets. The application may also provide a member name.

Table 36. SJ parameter list

Parameter	Usage	Format	Content
1	in	structure	'SJ', X'0001'
2	in/out	binary word	mtoken Must contain zero when the service is called. The service will supply a value if the service is successful.
3	in	vstring	DD name or path If the string data begins with a slash or a period followed by a slash it is treated as a path, otherwise as a DD name.
4 (optional)	in	vstring	Member name or path extension If parameter 3 is a DD name defining a PDSE or concatenation this is a member name and is required. If parameter 3 is either a path or a DD name defining a path, this parameter is optional and is appended to the path name if present.

Sample assembler code

```
         CALL  (15),(SJIL,MTOKEN,UNIXDD),VL
*        Note: 4th optional parameter omitted here

SJIL     DC    C'SJ',X'0001'
MTOKEN   DC    F'0'          Replaced by the service
UNIXDD   DC    H'5','UPATH'  Name of a DD statement,
*                            which in turn has PATH.
```

SQ - Starting a session with a CSVQUERY token

The calling application identifies a program that is currently loaded in virtual storage by providing a token returned from the MVS CSVQUERY service. This service cannot be used to access a program in LPA or LLA.

Note that the loaded program is only that portion of the program object required during execution. Stored program objects contain a good deal of information in addition to that required during execution. That additional information will be needed for the "Get" type services described later, so this service call, like the other Start service calls, will locate the program object on DASD and read it.

Table 37. SQ parameter list

Parameter	Usage	Format	Content
1	in	structure	'SQ', X'0001'

Table 37. SQ parameter list (continued)

Parameter	Usage	Format	Content
2	in/out	binary word	mtoken Must contain zero when the service is called. The service will supply a value if the service is successful.
3	in	2 words	eptoken An 8-byte field with content provided by the CSVQUERY service.

Sample assembler code

```
         CALL  (15),(SQIL,MTOKEN,CSVTOKEN),VL

SQIL     DC    C'SQ',X'0001'
MTOKEN   DC    F'0'          Replaced by the service
CSVTOKEN DS    D             Supplied by CSVQUERY
```

SS - Starting a session with a System DCB

Note: This service is currently treated as identical to SB and is retained for compatibility. See “SB - Starting a session with a BLDL identifier” on page 117 for more information and restrictions.

Table 38. SS parameter list

Parameter	Usage	Format	Content
1	in	structure	'SS', X'0001'
2	in/out	binary word	mtoken
3	in	structure	DCB
4	in	structure	Directory entry or DD name. Only the first 8 bytes are used.

GC - Getting Compile unit information

Compile unit information is primarily data passed to the binder by compilers, identifying the source of each program making up the program object being inspected. As an important special case, though, the first compile unit entry returned when the cursor is specified as zero provides information on the DASD location of the program object itself. The program object source "header record" returned by fast data in the CUI buffer for a program object in a PDSE will never identify the data set containing the object.

When no culist is provided, the cursor is an index into an ordered list of all CUI entries that can be returned. If the application does not modify the cursor during the retrieval process, multiple calls return all CUI records in the order by CU number because the buffer is full. When the culist is provided, the cursor is an index into that application-provided list. CUI records are returned in the order specified in the culist. If the application still does not modify the cursor during the retrieval process, multiple calls continue with subsequent entries in the list because the buffer is full. End of data is signalled when the end of the application-provided list is reached.

Table 39. GC parameter list

Parameter	Usage	Format	Content
1	in	structure	'GC', X'0001'
2	in	binary word	mtoken
3 (optional)	in	binary words	culist An array of numbers. The first word is the number of additional words that follow it. Each additional word is a compile unit number returned in BNL_SECT_CU by a 'GN' call. If no compile unit numbers are passed, the first (and only) word must be zero.
4	in/out	structure	buffer Must be a CUI buffer formatted by IEWBUFF or as defined in Appendix D, “Binder API buffer formats,” on page 239.
5	in/out	binary word	cursor Cursor is an index within the culist or into an ordered list of all CUI entries.
6	out	binary word	count The number of CUI records returned by the binder.

Sample assembler code

```
         CALL  (15),(GCIL,MTOKEN,NULL,BUFF,CURS,CNT),VL

GCIL     DC    C'GC',X'0001'
MTOKEN   DS    F             As set at Start call
NULL     DC    F'0'          Omitted to get all CU's
CURS     DC    F'0'          Start with PO information
CNT      DS    F             Number of records returned
BUFF  IEWBUFF  FUNC=MAPBUFF,TYPE=CUI,VERSION=6,SIZE=2000
```

GD - Getting Data from any class

This service is often used to access program text, but can also be used to retrieve data from other compiler-defined or binder-defined classes. The data can optionally be limited to that associated with a particular section.

One special feature for program text is that the addresses within the text can be relocated in the same way that the loader would do, relative to a specified starting address.

Note that many programs are built with multiple text classes. The GD service is not able to combine data from different classes in a single call, and the cursor used for positioning is always relative to the beginning of the specified class (or section contribution within the class if the optional section parameter is provided). Typically the calling application views program text as continuous across classes within a loadable segment. The application can adjust for this by using the class starting offset within the segment as returned by the GN service in the BNL_SEGM_OFF field.

Table 40. GD parameter list

Parameter	Usage	Format	Content
1	in	structure	'GD', X'0001'
2	in	binary word	mtoken
3	in	vstring	class
4 (optional)	in	vstring	section
5	in/out	structure	buffer Must be formatted by IEWBUFF or as defined in Appendix D, “Binder API buffer formats,” on page 239, and appropriate to the class requested.
6	in/out	binary word	cursor - in items The item size depends on the buffer type and buffer version used. For text data, this might not be returned as the next location after the last text byte returned, because pad bytes between sections and uninitialized data areas within sections may have been skipped. Any data skipped should be treated by the caller as containing the fill character (normally X'00').
7	out	binary word	count - in items
8 (optional)	in	64-bit address	relocation value An assumed address for the start of the class, to be used for address constant relocation.

Sample assembler code

```
         CALL  (15),(GDIL,MTOKEN,CLASS,,BUFF,CURS,CNT),VL

GDIL     DC    C'GD',X'0001'
MTOKEN   DS    F              As set at Start call
CLASS    DC    H'6',C'B_TEXT' One particular text class
CURS     DC    F'0'           Start at beginning of text
CNT      DS    F              Number of bytes returned
*                             (since text item size is 1)
BUFF  IEWBUFF  FUNC=MAPBUF,TYPE=TEXT,BYTES=8192
*        Note default to V1, but text buffer hasn't changed
```

GE - Getting External Symbol Dictionary data

This service is a specialized variant of GD, used to retrieve only one class, B_ESD. This can be a confusing class, though, because ESD records are owned by elements in a second class and may point to elements in a third class. For example, an ESD may describe an adcon in class C_CODE that refers to an address in class B_TEXT. Using the 'GD' service you would only be able to indicate that you were interested in class B_ESD, and would have to retrieve all ESDs to locate the specific ones you were interested in. The 'GE' service allows the caller to screen ESDs returned, limiting the output to those owned by a specified class. The calling application can also ask for ESDs in a specific section.

Table 41. GE parameter list

Parameter	Usage	Format	Content
1	in	structure	'GE', X'0001'
2	in	binary word	mtoken
3 (optional)	in	vstring	class
4 (optional)	in	vstring	section
5	in/out	structure	buffer Must be formatted by IEWBUFF or as defined in Appendix D, “Binder API buffer formats,” on page 239, and appropriate to the class requested.
6	in/out	binary word	cursor - in items The item size depends on the buffer type and buffer version used.
7	out	binary word	count - in items

Sample assembler code

```
         CALL  (15),(GEIL,MTOKEN,CLASS,,BUFF,CURS,CNT),VL

GEIL     DC    C'GE',X'0001'
MTOKEN   DS    F              As set at Start call
CLASS    DC    H'6',C'C_CODE' Limit ESDs returned
CURS     DC    F'0'           Start with first ESD
CNT      DS    F              Number of ESDs returned
BUFF IEWBUFF FUNC=MAPBUF,TYPE=ESD,VERSION=6,SIZE=50
*             i.e. a buffer big enough to hold 50 ESDs,
*             assuming the names are not too long.
```

GN - Getting Names of sections or classes

This is often the first Get service requested, because it provides the names of sections or classes within the program object, thus providing information required for the GD or GE service calls. If this service is called with no buffer it returns, in the count field, the number of classes or sections in the program object.

Table 42. GN parameter list

Parameter	Usage	Format	Content
1	in	structure	'GN', X'0001'
2	in	binary word	mtoken
3	in	character	ntype: C for class or S for section May be upper or lower case.
4 (optional)	in/out	vstring	buffer Must be an NAME buffer formatted by IEWBUFF or as defined in Appendix D, “Binder API buffer formats,” on page 239.
5	in/out	binary word	cursor - in items The item size depends on the buffer type and buffer version used.
6	out	binary word	count - in items

Sample assembler code

```
         CALL  (15),(GNIL,MTOKEN,TYPE,BUFF,CURS,CNT),VL

GNIL     DC    C'GN',X'0001'
MTOKEN   DS    F             As set at Start call
TYPE     DC    C'C'          To return class names/info
CURS     DC    F'0'          Start with first class
CNT      DC    F             Number of classes returned
BUFF IEWBUFF FUNC=MAPBUF,TYPE=NAME,VERSION=6,SIZE=50
*             i.e. a buffer big enough to hold 50 classes.
*             (Class names are never too long.)
```

RC - Return Code information

This service is useful when fast data services are called from a language which does not allow the calling application to inspect register 15 or register 0 on return from the service. It provides the return and reason codes from the most recent service call with the same mtoken.

Table 43. RC parameter list

Parameter	Usage	Format	Content
1	in	structure	'RC', X'0001'
2	in	binary word	mtoken
3	out	binary word	return code
4	out	binary word	reason code

EN - Ending a session

This service should always be issued after the calling application has finished requesting data services. It allows the fast data utility to clean up its resources.

Table 44. EN parameter list

Parameter	Usage	Format	Content
1	in	structure	'EN', X'0001'
2	in	binary word	mtoken

The Unitary interface

The Unitary interface is the original interface to fast data services. It continues to be supported, but IBM recommends using the Request Code interface instead. See “The Request Code interface” on page 114 for details about the Request Code interface.

The intent of this interface was to provide a single call to the service which could, in simple cases, retrieve all data required. To allow for the possibility that differing types of data may be required from a single program object, or that more data may be present than can fit in the caller's buffer, fast data Access service allows chaining of calls. A module token (MTOKEN) is created on the first call and returned to the caller. On the second and subsequent calls, the caller passes the MTOKEN to the service and is assured of continuing from the last call. On the last or only call of the set, HOLD=N should be specified, allowing all resources acquired by fast data access to be released.

Most significantly, the first call of a set has the overhead of reading the program object into virtual storage. This is required even if an eptoken for an already loaded program is provided, since the program object contains a good deal of additional data not present in a loaded copy of the executable text. A copy of the program object is retained in 64-bit storage until the call with HOLD=N is made.

The Unitary interface can be used either by making a call with a parameter list as described later or by using the assembler IEWBFDA macro which will generate the parameter list. If you intend to use a manually generated parameter list you should first understand the parameters as described for the macro.

See also “Parameter descriptions” on page 110. Those descriptions apply to all interfaces to fast data.

The IEWBFDA macro

The syntax of the IEWBFDA macro is:

```
[label]          IEWBFDA     ENTRYPNT=xentry point

                             ,MTOKEN=mtoken
                             ,RETCODE=retcode
                             ,RSNCODE=rsncode
                             {,EPTOKEN=eptoken
                              |  ,DDNAME=ddname[,MEMBER=member]
                              |  ,PATH=pathname
                              |  ,DCBPTR=dcbptr,DEPTR=deptr}
                             ,CLASS=class
                             [,SECTION=section]
                             ,AREA=buffer
                             ,CURSOR=cursor
                             ,COUNT=count
                             ,HOLD={N|Y}
                             ,SYSTEMDCB={N|Y}
                             [,DELETE={N|Y}]
                             [,LOADFAIL=loadfail]
                             ,MF={S | L | (E,listaddress)}
```

label
Optional symbol. If present, the label must begin in column 1.

ENTRYPNT=*entry_point* **– RX-type address or register (3-12)**
Specifies the name of a 4-byte variable that contains the entry point location of the IEWBFDAT module. If this location is known prior to the first call of the fast data access service, the use of such value by this parameter might save the implied processing overhead of attempting to load the module into storage.

MTOKEN=*token* **– RX-type address or register (3-12)**
Specifies the name of an 8-byte variable that contains the Module Token established by the binder. This token should be initialized to binary zero before the first call of the set.

RETCODE=*retcode* **– RX-type address or register (3-12)**
Specifies the name of a fullword integer variable that receives the completion code returned by the binder.

RSNCODE=*rsncode* **– RX-type address or register (3-12)**
Specifies the name of a 4-byte hexadecimal string variable that receives the reason code returned by the binder.

EPTOKEN=*eptoken* **– RX-type address or register (3-12)**
Specifies the name of an 8-byte variable that contains the EPTOKEN received from MVS contents supervisor by means of the CSVQUERY macro. EPTOKEN is required to locate the correct copy of a module in a PDSE program library, when the module has already been loaded into virtual storage by loader services.

DDNAME=*ddname* **– RX-type address or register (3-12)**
Specifies the name of a structure identifying the ddname of the library to be accessed. The structure consists of a two-byte length field followed by a character string of up to 8 bytes. The library identified by the ddname must not include a member name.

DDNAME may also point to a z/OS UNIX System Services file by using a path specification for the ddname. In this case, no member name should be passed to IEWBFDAT.

PATH=*pathname* **– RX-type address or register (3–12)**
An absolute or relative path name may be passed explicitly. The path name will occupy the same parameter list position as DDNAME and must start with **/** or **./**.

DCBPTR=*dcbptr* **– RX-type address or register (3-12)**
Specifies the name of a 4-byte pointer variable containing the address of a DCB that represents a PDSE program library. The DEPTR parameter is required if DCBPTR is specified.

Notes:

1. The DCB must be opened before the invocation of the service, and the DCB parameters must be DSORG=PO,MACRF=(R). See “SB - Starting a session with a BLDL identifier” on page 117 for a listing of the restrictions on DCBs.
2. If you haven't set a new DCBE with the EADSCB=OK option, the system assumes that the program cannot handle the track address and issues a new ABEND.

DEPTR=*deptr* **– RX-type address or register (3-12)**
Specifies the name of a 4-byte pointer variable containing the address of a directory entry as returned by BLDL.

MEMBER=*member* **– RX-type address or register (3-12)**
Specifies the name of a structure identifying the member name or alias of the library member to be accessed. The structure consists of a two-byte length field followed by a character string of up to 1024 bytes. MEMBER may be specified only if DDNAME is specified.

SECTION=*section* **– RX-type address or register (3-12)**
Specifies the name of a structure identifying the name of the section to be accessed. The structure consists of a two-byte length field followed by a character string of up to 32767 bytes. This is an optional parameter, and is default to a concatenation of all sections in the specified class.

CLASS=*class* **– RX-type address or register (3-12)**
Specifies the name of a structure identifying the class name of the required class. The structure consists of a two-byte length field followed by a character string of up to 16 bytes.

AREA=*buffer* **– RX-type address or register (3-12)**
Specifies the name of a standard buffer that receives the data. See Appendix D, “Binder API buffer formats,” on page 239 for buffer formats. The version level is specified with this keyword.

CURSOR=*cursor* **– RX-type address or register (3-12)**
Specifies the name of a fullword integer variable that indicates to the binder the position (relative record or byte) in the buffer where processing is to begin.

The CURSOR value identifies an index into the requested data beginning with 0 for the first data item. Each of the buffer formats defined in Appendix D, “Binder API buffer formats,” on page 239 contains an entry length field in its header. Multiplying the cursor value by the entry length provides a byte offset into the data. Note that CUI, LIB, PMAR, and text data is always treated as having entry length 1.

The CURSOR value is an input and output parameter. On input to the service, the cursor specifies the first item to return. On exit from the service, it is updated to an appropriate value for continued sequential retrieval if not all data has yet been retrieved. For text data this may or may not be the next byte after the last one returned, because pad bytes between sections and uninitialized data areas within sections may have been skipped.

COUNT=*count* **– RX-type address or register (3-12)**
Specifies the name of a fullword integer variable in which the binder can store the number of bytes or entries returned in the buffer.

HOLD=N|Y – RX-type address or register (3-12)
Points to a 1 byte field containing a **Y** or **N**. If Yes is specified, DIV maps and acquired storage are not released, allowing requests for additional data from the same module on subsequent calls.

SYSTEMDCB=N|Y – RX-type address or register (3-12)
Points to a 1 byte field containing a **Y** or **N**, with a default of **N**. This parameter is effectively ignored by fast data, but retained for compatability.

DELETE=N|Y – RX-type address or register (3-12)
Specifies if the IEWBFDA module is deleted The default is 'Y'.

If Yes is specified, or if HOLD=N is specified without DELETE being specified, the module is deleted.

If Yes is specified, or if DELETE is not specified but HOLD=Y is specified, the module is not deleted.

LOADFAIL=*loadfail* **– RX-type address or register (3-12)**
Specifies the name of a four-byte pointer containing the address of a user-defined routine that gets control if the LOAD operation fails while attempting to load the IEWBFDA module.

MF={S | L | (E,*listaddress***)}**
Specifies the macro form required.

S Specifies the standard form and generates a complete inline expansion of the parameter list. The standard form is for programs that are not reenterable or refreshable, and for programs that do not change values in the parameter list. The standard form is the default.

L Specifies the list form of the macro. This form generates an inline parameter list.

E Specifies the execute form of the macro. This form updates a parameter and transfers control to the service routine.

listaddress **– RX-type address or register (3-12)**
Specifies the address of the parameter list.

Unitary parameter list

If you would like to use the Unitary interface without using the macro, you must load IEWBFDAT and call using standard MVS linkage conventions, passing the following parameter list.

The address of the following parameter list must be passed in GPR 1:

Table 45. IEWBFDA parameter list

```
PARMLIST        DS      0F
                DC      A(ENTRYPNT)          Module entry point
                DC      A(MTOKEN)            Module token
                DC      A(RETCODE)           Return code
                DC      A(RSNCODE)           Reason code
                DC      A(EPTOKEN)           EPTOKEN
                DC      A(DDNAME | PATH)     DD name or path name
                DC      A(MEMBER)            Member name
                DC      A(DCBPTR)            DCB pointer
                DC      A(DEPTR)             Directory entry pointer
                DC      A(CLASS)             Class name
                DC      A(SECTION)           Section name
                DC      A(AREA)              Standard data buffer
                DC      A(CURSOR)            Starting position
                DC      A(COUNT)             Data count
                DC      A(HOLD)              Hold flag
                DC      A(SYSTEMDCB)         SystemDCB indicator
*               Keyword Values
HOLD            DC      CL1'Y'               Use N on the last or only request.
SYSTEMDCB       DC      CL1'Y'               Indicates whether passed DCB is from a
                                             system library.

                                                   'Y' = YES
                                                   'N' = NO default
```

Note: SYSTEMDCB is an optional positional parameter which, if specified, follows the HOLD parameter. You must add a value of X'80000000' to the *last* parameter you specify – either the HOLD parameter (if you do not specify SYSTEMDCB) or, if you specify it, the SYSTEMDCB parameter.

Error handling

Fast data access does not provide error handling routines for logic failures caused by input or environmental problems. It does use error returns from system services where those are reasonably available, and reflects such errors in the reason codes it returns.

In a few cases fast data writes diagnostic messages to the job log (JESYSMG). These have IEW2*nnn* message numbers, and are documented in the appropriate volume of *z/OS MVS System Messages*.

If a problem needs to be reported to IBM, the following will be helpful:

- For an ABEND or program check in fast data, a SLIP dump or SYSMDUMP which includes 64-bit storage.
- For an unexpected return and reason code, a trace which can be gotten by allocating an IEWTRACE DD statement. fast data uses QSAM to write variable blocked records to the data set or SYSOUT destination specified.

Return and reason codes

Return Code	Reason Code	Explanation
00	00000000	Normal completion.
04	10800001	Normal completion. Data was returned, and the buffer is full. There might be additional data to retrieve.
04	10800002	Normal completion. Some data might have been returned in the buffer, and an end-of-data condition was encountered.
08	10800010	The requested CLASS does not exist in the indicated program object. No data was returned.
08	10800011	The requested SECTION does not exist in the indicated program object. The request was rejected.
08	10800061	The third parameter for the GN service must contain EBCDIC 'C' or 'S' in upper or lower case.
08	10800062	No sections were found.
08	10800064	The culist passed to the GC service contained a value which had not been returned by the GN service.
12	10800020	The program object identified by EPTOKEN could not be located in physical storage. The request was rejected.
12	10800021	The specified MEMBER could not be found. The request was rejected.
12	10800022	The buffer provided (AREA) is too small to contain even one record of the specified CLASS. The request was rejected.
12	10800023	A required parameter was omitted or specified as zero.
12	10800024	An incorrect module token (MTOKEN) was provided. The request was rejected.
12	10800025	A buffer (AREA) with a negative length was provided. The request was rejected.
12	10800026	The provided CLASS name is not valid. The request was rejected.
12	10800027	The provided CURSOR value is either negative, or, if this was a first or a subsequent call, it was altered by the caller of the service so that the (new) value was not within the range of records for the SECTION being processed. The request was rejected.
12	10800028	The provided HOLD value is not one of the allowed values: 'Y' or 'N'. The request was rejected.
12	10800029	The program is not a program object, anomalies were found in its structure, or the program is PO1 (program object, version 1) and the program contains overlay structures. The request was rejected.
12	1080002A	The program object identified by the EPTOKEN has been updated since it was last fetched. The service cannot process it, since the extracted data would not be the same as the data in physical storage. The request was rejected.
12	1080002B	The program object identified by the EPTOKEN was fetched by a product other than DFP and therefore it cannot be processed. The request was rejected.

Return Code	Reason Code	Explanation
12	1080002C	The data set identified by DDNAME could not be found, or the system service (SVC99) encountered an error while verifying the data set allocation. The request was rejected.
12	1080002D	The specified DDNAME did not refer to a program object library. The request was rejected.
12	1080002E	An error occurred while trying to open the data set identified by DDNAME. The request was rejected.
12	1080002F	The acquisition of working storage was not successful. The request was rejected.
12	10800030	An error was encountered while attempting to verify the provided EPTOKEN. (Service Involved: CSVQUERY). The request was rejected.
12	10800031	An error was encountered while attempting to map the program object onto a data space. The request was rejected.
12	10800032	An error was encountered while attempting to access the directory entry for the program object being processed. (Service Involved: DESERV). The request was rejected.
12	10800036	The supplied DCB, identified by DCBPTR, was not OPENed prior to the service's invocation as required. The request was rejected.
12	10800037	The supplied DCB, identified by DCBPTR, is OPEN but one or both of the pertinent DCB parameters are not DSORG=PO and MACRF=(R), as required. The request was rejected.
12	10800038	The supplied directory entry, identified by DEPTR, does not represent a program object. The request was rejected.
12	10800039	The supplied buffer (AREA) has an incorrect version number in its header, which disagrees with the version of the program object being accessed. For example, the buffer version is 1, but the program object is version 2.
12	1080003A	The supplied buffer (AREA) has an incorrect class name in its header, which disagrees with CLASS name passed to the service.
12	1080003B	The provided SYSTEMDCB parameter is not one of the allowed values: Y or N, in uppercase or lowercase EBCDIC.
12	1080003D	The directory portion of the supplied path name is invalid.
12	1080003F	The supplied path name is invalid.
12	10800040	The supplied path name exceeds the acceptable length limit.
12	10800041	Read access error occurred trying to open the file supplied on the PATH parameter.
12	10800042	Read error occurred trying to open the file supplied on the PATH parameter.
12	10800063	Interface value is invalid.

Return Code	Reason Code	Explanation
12	10800065	The cursor passed on a GD request for class B_PARTINIT is invalid. The cursor must point to the beginning of a part initializer record.

Chapter 5. IEWBNDD, IEWBNDDX - Binder C/C++ API DLL functions

This topic describes the binder C/C++ API that enables C/C++ programs to interface easily with binder and to get the similar functionality of the binder API described in "IEWBIND function reference" on page 24 and fast data access services described in Chapter 4, "IEWBFDAT - Binder Fast data access API functions," on page 109. The previously defined APIs depend on buffer structures with assembler mappings described in Chapter 2, "IEWBUFF - Binder API buffers interface assembler macro for generating and mapping data areas," on page 13. The APIs can be called from other languages, but the coding may be awkward. The programmer is responsible for providing a certain environment, buffers and parameters lists. The binder C/C++ API is a C code interface that implements the initialization and access functions of the binder API and fast data access services. The interface provides C functions equivalent to the following binder API macro functions and fast data access services via request code call interface:

Binder API	Fast data access
"ADDA: Add alias" on page 28	SB- Start with DCB
"ALIGNT: Align text" on page 30	SJ – Start with DDname
"ALTERW: Alter workmod" on page 32	SQ – Start with EPtoken
"AUTOC: Perform incremental autocall" on page 36	SS – Start with system DCB
"BINDW: Bind workmod" on page 37	GC – Get compile unit list
"CREATEW: Create workmod" on page 41	GD – Get data
"DELETEW: Delete workmod" on page 42	GE – Get ESD data
"DLLR: Rename DLL modules" on page 44	GN – Get names
"ENDD: End dialog" on page 46	EN – End a session
"GETC: Get compile unit list" on page 47	
"GETD: Get data" on page 50	
"GETE: Get ESD data" on page 52	
"GETN: Get names" on page 56	
"IMPORT: Import a function or external variable" on page 59	
"INCLUDE: Include module" on page 61	
"INSERTS: Insert section" on page 68	
"LOADW: Load workmod" on page 69	
"ORDERS: Order sections" on page 72	
"PUTD: Put data" on page 73	
"RENAME: Rename symbolic references" on page 78	
"RESETW: Reset workmod" on page 79	
"SAVEW: Save workmod" on page 81	
"SETL: Set library" on page 86	

© Copyright IBM Corp. 1991, 2012

Binder API	Fast data access
"SETO: Set option" on page 88	
"STARTD: Start dialog" on page 91	
"STARTS: Start segment" on page 98	

The interface also provides some utilities functions to allow users to access return values and enables the creation of binder API buffers for some API calls that require buffers as input.

Using the binder C/C++ API and headers

The binder C/C++ API provides a header file (__iew_api.h), which allows C/C++ programs to interface with binder API and fast data access services more naturally in a C/C++ environment. The header file includes the following:

- API macros

  The following macros are used to control the behavior of C/C++ APIs:

  _IEW_TARGET_RELEASE

  This feature test macro is used to control the expansion of the header file (__iew_api.h). To use the macro, define it to the desired value before you define the #include <__iew_api.h>. Declarations made by __iew_api.h correspond to the function available with the specified release. The following macros are valid values for _IEW_TARGET_RELEASE:

  – _IEW_ZOSV1R9_
  – _IEW_ZOSV1R10_
  – _IEW_ZOSV1R11_
  – _IEW_ZOSV1R12_
  – _IEW_ZOSV1R13_
  – _IEW_CURRENT_

  The default value is defined to the _IEW_ZOSV1R9_ macro.

  It is suggested that you use the _IEW_TARGET_RELEASE macro as the __release value with both the __iew_openW() and the __iew_fd_open() functions. This feature test macro is always a valid value for _IEWTargetRelease. Using the feature test macro with these APIs ensures that values returned by all the APIs are consistent with the declarations made by the __iew_api.h header.

  Notes:

  1. If you do not define the feature test macro to _IEW_ZOSV1R9_, _IEW_ZOSV1R10_, _IEW_ZOSV1R11_, _IEW_ZOSV1R12_ or _IEW_ZOSV1R13_, a compile-time error occurs.
  2. If you do not use the same value of the feature test macro when defining applications, parts of the application might incorrectly interpret the results of some API calls.

  _IEW_ZOSV1R9_

  This macro is used to identify the release z/OS V1.9. Use the macro with the feature test macro _IEW_TARGET_RELEASE.

  _IEW_ZOSV1R10_

  This macro is used to identify the release z/OS V1.10. Use the macro with the feature test macro _IEW_TARGET_RELEASE.

_IEW_ZOSV1R11_
This macro is used to identify the release z/OS V1.11. Use the macro with the feature test macro _IEW_TARGET_RELEASE.

_IEW_ZOSV1R12_
This macro is used to identify the release z/OS V1.12. Use the macro with the feature test macro _IEW_TARGET_RELEASE.

_IEW_ZOSV1R13
This macro is used to identify the release z/OS V1.13. Use the macro with the feature test macro _IEW_TARGET_RELEASE.

_IEW_CURRENT_
This macro is used to identify the current z/OS release. Use the macro with the feature test macro _IEW_TARGET_RELEASE. By using this value, you can always use the most current binder capabilities.

Note: All the parts that comprise an application must have the same value for _IEW_TARGET_RELEASE. If you define _IEW_TARGET_RELEASE to _IEW_CURRENT_, and subsequently the system is upgraded to a new z/OS release, the value of _IEW_CURRENT_ might different than it was in the prior z/OS release. It is not necessary to recompile the application, the application continues to operate using the _IEW_TARGET_RELEASE at which it was compiled. However, if you want to recompile any parts of the application, you must recompile all parts of the application to ensure that all the parts in the application use the same _IEW_TARGET_RELEASE.

- API data types

  The API data types are used by API access and utilities functions. They allow for data encapsulation and as a convenient way for parameters passing.

- Buffer header and entry definition for each buffer type

  The buffer header and buffer entry definition for each buffer type are based on the version 6 and later API buffer formats as described in Appendix D, “Binder API buffer formats,” on page 239.

- Binder API access functions

```
__iew_addA()          __iew_alignT()          __iew_alterW()
__iew_autoC()         __iew_bindW()           __iew_closeW()
__iew_getC()          __iew_getD()            __iew_getE()
__iew_getN()          __iew_import()          __iew_includeName()
__iew_includePtr()    __iew_includeSmde()     __iew_includeToken()
__iew_insertS()       __iew_loadW()           __iew_openW()
__iew_orderS()        __iew_putD()            __iew_rename()
__iew_resetW()        __iew_saveW()           __iew_setL()
__iew_setO()          __iew_startS()
```

- Binder API utilities functions

```
__iew_api_name_to_str()  __iew_create_list()       __iew_eod()
__iew_get_reason_code()  __iew_get_return_code()  __iew_get_cursor()
__iew_set_cursor()
```

- Fast data access functions

```
__iew_fd_end()         __iew_fd_getC()           __iew_fd_getD()
__iew_fd_getE()        __iew_fd_getN()          __iew_fd_open()
__iew_fd_startDcb()    __iew_fd_startDcbS()      __iew_fd_startName()
__iew_fd_startToken()
```

- Fast data utilities functions

```
__iew_fd_eod()                  __iew_fd_get_reason_code()
__iew_fd_get_return_code()     __iew_fd_get_cursor()
__iew_fd_set_cursor()
```

C/C++ header file __iew_modmap.h declares the structures which are used in the binder module map IEWBMMP. IEWBMMP is usually produced by using the binder MODMAP option, and it is described in"Module Map" on page 308. This header allows a C/C++ program to more easily reference the module map (whether reading it directly or by using the binder APIs).

Accessing the binder C/C++ API is through DLL support and is packaged as follows:

Table 46. Accessing binder C/C++ API through DLL support

side-deck name	DLL name	DLL type	side-deck location	DLL location
iewbndd.x	iewbndd.so	31-bit non-XPLINK	/usr/lib (UNIX filesystem)	/usr/lib (UNIX filesystem)
iewbnddx.x	iewbnddx.so	31-bit XPLINK	/usr/lib (UNIX filesystem)	/usr/lib (UNIX filesystem)
IEWBNDD	IEWBNDD	31-bit non-XPLINK	SYS1.SIEASID (MVS data set)	SYS1.SIEAMIGE (MVS data set)
IEWBNDDX	IEWBNDDX	31-bit XPLINK	SYS1.SIEASID (MVS data set)	SYS1.SIEAMIGE (MVS data set)

These side-deck and DLL pairs must be used together. For example, since SYS1.SIEASID(IEWBNDDX) corresponds to DLL name IEWBNDDX, if your application is bound with this side-deck it will use the DLL named IEWBNDDX, which is installed into SYS1.SIEAMIGE.

Refer to *z/OS Language Environment Programming Guide* for rules on how DLLs are located and loaded from different locations. Also, since SYS1.SIEAMIGE is automatically part of the MVS LNKLST, there is normally no need to explicitly set the search order. On the other hand, while IBM recommends that /usr/lib be on the LIBPATH environment variable via /etc/profile, it is not required. Nor is LIBPATH set by default for programs executed where the login shell has not been run (such as from batch).

Notes:

1. XPLINK version of binder C/C++ API through DLL support can improve performance for XPLINK callers of binder C/C++ API.
2. __iew_api_to_name(), listed above for the binder API, can also be used for the fast data API.

Environment

The environment is the same as described in either Chapter 1, "Using the binder application programming interfaces (APIs)," on page 1 for binder API users or Chapter 4, "IEWBFDAT - Binder Fast data access API functions," on page 109 for fast data users.

Binder API functions

This topic describes the binder API functions.

__iew_addA() – Add alias

Adds a new alias to the list of aliases for a program, or changes the AMODE of an entry point. Can be used when binding or copying a program.

Format

```
#include <__iew_api.h>
int __iew_addA(_IEWAPIContext *__context,
                           const char *__aname, const char *__ename,
                           _IEWAnameType __anametype,
                           const char *__amode);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__aname
the name of the alias to be added

__ename
existing external symbol in the program to be associated with the alias

__anametype
alias type can be one of the following:

_IEW_ALIAS
_IEW_SYM_LINK
_IEW_PATH

__amode
amode and the values that can be specified for amode are ‘24’, ‘31’, ‘64’, ‘ANY’, and ‘MIN’

Returned Value

If successful, __iew_addA() returns 0.

If unsuccessful, __iew_addA() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_get_return_code().

Utilities Functions

__iew_get_reason_code()
__iew_get_return_code()

__iew_alignT() – Align text

Requests that a specified section be loaded on a 4K boundary. Can be used only when _IEW_BIND intent is specified.

Format

```
#include <__iew_api.h>
int __iew_alignT(_IEWAPIContext *__context,
                           const char *__section);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__section
the section name

Returned Value

If successful, __iew_alignT() returns 0.

If unsuccessful, __iew_alignT() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_get_return_code().

Utilities Functions

__iew_get_reason_code()
__iew_get_return_code()

__iew_alterW() – Alter workmod

Changes or deletes symbols or sections, or adds patch space to a control section. Can be used only when _IEW_BIND intent was specified.

Format

```
#include <__iew_api.h>
int __iew_alterW(_IEWAPIContext *__context,
                             _IEWAltType __alttype, _IEWAltMode __altmode,
                             const char *__oldname, const char *__newname,
                             unsigned int *__count,
                             const char *__class);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__alttype
alter type can be one of the following:

- _IEW_ALT_CHANGE
  changes an external symbol of any ESD type from old name to new name
- _IEW_ALT_DELETE
  deletes an external symbol in the target modules
- _IEW_ALT_EXPAND
  expands the length of the text of a section
- _IEW_ALT_REPLACE
  deletes a symbol (old name), and changse any references to that symbol to a new name

__altmode
alter mode can be one of the following:

- _IEW_ALT_NEXT
  applies only to what will be in the next module to be included into the workmod

- _IEW_ALT_IMMED
  applies to what is already in the workmod

__oldname
the old name

__newname
the new name

__count
the number of bytes

__class
the class name

Returned Value
If successful, __iew_alterW() returns 0.

If unsuccessful, __iew_alterW() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_get_return_code().

Utilities Functions
__iew_get_reason_code()
__iew_get_return_code()

__iew_autoC() – Perform incremental autocall
Attempts to resolve currently unresolved symbols using the contents of the specified library. Can be used only when _IEW_BIND intent was specified.

Format
```
#include <__iew_api.h>
int __iew_autoC(_IEWAPIContext *__context,
                           const char *__dd_or_path);
```

Parameters Descriptions
__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__dd_or_path
the autocall library DD name or path name

Returned Value
If successful, __iew_autoC() returns 0.

If unsuccessful, __iew_autoC() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_get_return_code().

Utilities Functions
__iew_get_reason_code()
__iew_get_return_code()

__iew_bindW() – Bind workmod

Resolves all external references if it is possible, computes relative location of all sections, builds relocation and external symbol dictionaries. Required when _IEW_BIND intent was specified, cannot be used otherwise.

Format

```
#include <__iew_api.h>
int __iew_bindW(_IEWAPIContext *__context,
                            const char *__ddname);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__ddname
an optional additional library to use for resolving symbols.

Returned Value

If successful, __iew_bindW() returns 0.

If unsuccessful, __iew_bindW() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_get_return_code().

Utilities Functions

__iew_get_reason_code()
__iew_get_return_code()

__iew_closeW() – Close workmod

The closeW() deletes all data associated within the context(_IEWAPIContext), releases IEWBIND from memory, and deletes the context.

Format

```
#include <__iew_api.h>
_IEWAPIContext  *__iew_closeW(_IEWAPIContext *__context,
                                    _IEWAPIFlags __flags,
                                    unsigned int *__return_code,
                                    unsigned int *__reason_code);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__flags
API flags protect whether to allow the binder to delete a workmod that has been altered but not yet saved or loaded.

Note: Set the flag to 1 to turn on the bit and 0 to turn off the bit.

__return_code
return code is passed back from DELETEW or ENDD

__reason_code
reason code is passed back from DELETEW or ENDD

Returned Value

If successful, __iew_closeW() returns NULL.

If unsuccessful, __iew_closeW() returns API context.

Note: If an invalid context is passed, __iew_closeW() returns NULL.

__iew_getC() – Get compile unit list

Obtains source information about the program and the compile units from which it was constructed.

Format

```
#include <__iew_api.h>
int __iew_getC(_IEWAPIContext *__context,
                          int __culist[],
                          _IEWCUIEntry ** __cui_entry);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__culist
compile unit list – array of compile units where the first entry is used to specify the total number of compile unit entries. If the first entry is non zero, then one record for each compile unit in a list of compile units will be returned. If the first entry is zero, then one record of each of all compile units will be returned.

__cui_entry
returned buffer – CUI entry, one record for each compile unit in a list of compile units is returned.

Returned Value

If successful, __iew_getC() returns > 0 (count, number of entries returned in the buffer).

If unsuccessful, __iew_getC() returns 0.

Utilities Functions

__iew_eod()
__iew_get_reason_code()
__iew_get_return_code()
__iew_get_cursor()
__iew_set_cursor()

__iew_getD() – Get data

Returns data associated with a specified class (and optionally section) in the program.

Format

```
#include <__iew_api.h>
int __iew_getD(_IEWAPIContext *__context,
                          const char *__class, const char * __sect,
                          long long * __reloc,
                          void ** __data_entry);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__class
class name.

__sect
section name.

__reloc
relocation address.

__data_entry
returned buffer – based on class name.

Returned Value

If successful, __iew_getD() returns a number greater than zero representing the number of data items or bytes returned in the buffer.

If unsuccessful, __iew_getD() returns 0.

Utilities Functions

__iew_api_name_to_str()
__iew_eod()
__iew_get_reason_code()
__iew_get_return_code()
__iew_get_cursor()
__iew_set_cursor()

__iew_getE() – Get ESD data

Returns external symbol dictionary information selected by various criteria.

Format

```
#include <__iew_api.h>
int __iew_getE(_IEWAPIContext *__context,
                        const char *__sect, const char *__class,
                        const char *__sym, const char *__rec_type
                        int *__offset,
                        _IEWESDEntry ** __esd_entry);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__sect
section name.

__class
class name. See class under “Understanding binder programming concepts” on page 1 for details.

__sym
symbol name. See “External symbol dictionary” in Chapter 2 of Program Management User’s Guide and Reference for details.

__rec_type
ESD record type.

__offset
offset in module or section

__esd_entry
returned buffer – ESD entry

Returned Value
If successful, __iew_getE() returns > 0 (count, number of entries returned in the buffer).

If unsuccessful, __iew_getE() returns 0.

Utilities Functions
__iew_api_name_to_str()
__iew_eod()
__iew_get_reason_code()
__iew_get_return_code()
__iew_get_cursor()
__iew_set_cursor()

__iew_getN() – Get names
Returns a list of section or class names within the program together with information about the sections or classes.

Format
```
#include <__iew_api.h>
int __iew_getD(_IEWAPIContext *__context,
                          _IEWNameType __nametype,
                          unsigned int *__total_count,
                          _IEWNameListEntry ** __name_entry);
```

Parameters Descriptions
__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__nametype
name type could be one of the following: _IEW_SECTION or _IEW_CLASS.

__total_count
total number of sections or classes in the workmod.

__name_entry
returned buffer – binder name list.

Returned Value
If successful, __iew_getN() returns > 0 (count, number of entries returned in the buffer).

If unsuccessful, __iew_getN() returns 0.

Utilities Functions
__iew_eod()
__iew_get_reason_code()
__iew_get_return_code()
__iew_get_cursor()
__iew_set_cursor()

__iew_import() – Import a function or external variable

Identifies a function or variable to be resolved dynamically during execution. Can be used only when _IEW_BIND intent is specified.

Format

```
#include <__iew_api.h>
int __iew_getD(_IEWAPIContext *__context,
                            _IEWImpType __impType,
                            const char *__dllname, const char *__iname,
                            unsigned int *__offset);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__impType
import type can be one of the following:

- _IEW_IMP_CODE
- _IEW_IMP_DATA
- _IEW_IMP_CODE64
- _IEW_IMP_DATA64

__dllname
DLL name.

__iname
symbol name.

__offset
function descriptor offset.

Returned Value

If successful, __iew_import() returns 0.

If unsuccessful, __iew_import() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_get_return_code().

Utilities Functions

__iew_get_reason_code()
__iew_get_return_code()

__iew_includeDeptr() – Include module via DEPTR

Add an object to the workmod identified by a DDname and BLDL list entry.

Format

```
#include <__iew_api.h>
int __iew_includeDeptr(_IEWAPIContext *__context,
                                        const char *__ddname,
                                        void *__deptr,
                                        _IEWAPIFlags *__flags);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__ddname
DD name.

__deptr
member entry returned by BLDL, at least 62 bytes long.

__flags
API flags

- __aliases - whether to include the program module aliases with the program module
- __attrib - whether to include the program module attributes with the program module
- __imports - whether or not the import statements are to be included from the input module

Note: Set the flag to 1 to turn the bit on or 0 to turn the bit off.

Returned Value

If successful, __iew_includeDeptr() returns 0.

If unsuccessful, __iew_includeDeptr() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_get_return_code().

Utilities Functions

__iew_get_reason_code()
__iew_get_return_code()

__iew_includeName() – Include module via NAME

Adds an object to the wokmod identified either by a path name or by a DDname and member name.

Format

```
#include <__iew_api.h>
int __iew_includeName(_IEWAPIContext *__context,
                                      const char *__dd_or_path,
                                      const char * __member,
                                      _IEWAPIFlags *__flags);
                                     _IEWAPIFlags *__flags);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__dd_or_path
DD name or pathname.

__member
member name if using DD name.

__flags
API flags

- __aliases - whether to include the program module aliases with the program module
- __attrib - whether to include the program module attributes with the program module
- __imports - whether or not the import statements are to be included from the input module

Note: Set the flag to 1 to turn the bit on or 0 to turn the bit off.

Returned Value

If successful, __iew_includeName() returns 0.

If unsuccessful, __iew_includeName() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_get_return_code().

Utilities Functions

__iew_get_reason_code()
__iew_get_return_code()

__iew_includePtr() – Include module via POINTER

Adds an object to the workmod identified by a DCB and BLDL list entry.

Format

```
#include <__iew_api.h>
int __iew_includePtr(_IEWAPIContext *__context,
                                    void  *__dcbptr,
                                    void *__deptr,
                                    _IEWAPIFlags *__flags);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__dcbptr
pointer to DCB.

__deptr
pointer to BLDL entry.

Returned Value

If successful, __iew_includePtr() returns 0.

If unsuccessful, __iew_includePtr() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_get_return_code().

Utilities Functions

__iew_get_reason_code()
__iew_get_return_code()

__iew_includeSmde() – Include module via SMDE

Adds an object to the workmod identified by a DDname and SMDE returned by DESERV.

Format

```
#include <__iew_api.h>
int __iew_includeSmde(_IEWAPIContext *__context,
                                        const char *__ddname,
                                        void *__deptr,
                                        _IEWAPIFlags *__flags);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__ddname
DD name.

__deptr
pointer to BLDL entry, the directory entry is in SMDE format.

__flags
API flags

- __aliases - whether to include the program module aliases with the program module
- __attrib - whether to include the program module attributes with the program module
- __imports - whether or not the import statements are to be included from the input module

Note: Set the flag to 1 to turn the bit on or 0 to turn the bit off.

Returned Value

If successful, __iew_includeSmde() returns 0.

If unsuccessful, __iew_includeSmde() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_get_return_code().

Utilities Functions

__iew_get_reason_code()
__iew_get_return_code()

__iew_includeToken() – Include module via DEPTR

Add an object to the workmod identified by an entry point token returned by CSVQUERY.

Format

```
#include <__iew_api.h>
int __iew_includeToken(_IEWAPIContext *__context,
                                        void *__eptoken,
                                        _IEWAPIFlags *__flags);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__eptoken
EPTOKEN, token from the CSVQUERY macro.

__flags
API flags

- __aliases - whether to include the program module aliases with the program module
- __attrib - whether to include the program module attributes with the program module
- __imports - whether or not the import statements are to be included from the input module

Note: Set the flag to 1 to turn the bit on or 0 to turn the bit off.

Returned Value

If successful, __iew_includeToken() returns 0.

If unsuccessful, __iew_includeToken() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_get_return_code().

Utilities Functions

__iew_get_reason_code()
__iew_get_return_code()

__iew_insertS() – Insert section

Position a control section or common area within the program, primarily for overlay structured program. Can be used only when _IEW_BIND intent was specified.

Format

```
#include <__iew_api.h>
int __iew_insertS(_IEWAPIContext *__context,
                            const char *__section);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__section
section name.

Returned Value

If successful, __iew_insertS() returns 0.

If unsuccessful, __iew_insertS() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_get_return_code().

Utilities Functions
__iew_get_reason_code()
__iew_get_return_code()

__iew_loadW() – Load workmod

Load the program into virtual storage for immediate execution.

Format
```
#include <__iew_api.h>
int __iew_loadW(_IEWAPIContext *__context,
                              _IEWAPIFlags __flags,
                              void *__eploc, const char *__lname,
                              _IEWExtentListEntry  **__xtlst_entry);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__flags
API flags:

- __identify - whether or not the loaded program module is identified to the system

Note: Set the flag to 1 to turn the bit on or 0 to turn the bit off.

__eploc
entry point address.

__lname
name used for identify.

__xtlst_entry
returned buffer – extent list.

Returned Value
If successful, __iew_loadW() returns 0.

If unsuccessful, __iew_loadW() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_get_return_code().

Utilities Functions
__iew_get_reason_code()
__iew_get_return_code()

__iew_openW() – Open workmod

The __iew_openW() creates a context(_IEWAPIContext) and initializes all of the parameters in the context and loads IEWBIND into memory. If successful, the context is created and returned to user for all subsequent API calls.

This is a ‘C’ function equivalent to binder API STARTD and CREATEW.

Format
```
#include <__iew_api.h>
_IEWAPIContext *__iew_openW(_IEWTargetRelease __release,
                                          _IEWIntent __intent,
```

```
                                        _IEWList *__file_list,
                                        _IEWList *__exit_list,
                                        const char *__parms,
                                        unsigned int *__return_code,
                                        unsigned int *__reason_code);
```

Parameters Descriptions

__release
target release can be one of the following:

- _IEW_ZOSV1R9
- _IEW_ZOSV1R10
- _IEW_ZOSV1R11
- _IEW_ZOSV1R12
- _IEW_ZOSV1R13

Note: It is recommended that you define the feature test macro _IEW_TARGET_RELEASE prior to the #include <__iew_api.h>, and that _IEW_TARGET_RELEASE be used for __release. This definition ensures that structure mappings in <__iew_api.h> are consistent with the data returned by the other API access functions.

__intent
processing intent can be one of the following:

- _IEW_ACCESS
- _IEW_BIND

Note: _IEW_ACCESS is more efficient but only allows the program to be inspected or copied. This can be changed later by __iew_resetW().

__file_list
file list is created by __iew_create_list().

__exit_list
exit list is created by __iew_create_list().

__parms
parameters – list of binder option.

__return_code
return code is passed back from STARTD or CREATEW.

__reason_code
reason code is passed back from STARTD or CREATEW

Returned Value

If successful, __iew_openW() returns API context.

If unsuccessful, __iew_openW() returns NULL.

Note: The returned value is the same as the code returned by a subsequent __iew_get_return_code().

Utilities Functions

__iew_create_list()

__iew_orderS() – Order sections

Specify the position of a section within the program. Normally a series of calls would be made listing sections in the desired order. Can be used only when _IEW_BIND intent was specified.

Format

```
#include <__iew_api.h>
int __iew_orderS(_IEWAPIContext *__context,
                              const char *__section);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__section
section name.

Returned Value

If successful, __iew_orderS() returns 0.

If unsuccessful, __iew_orderS() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_get_return_code().

Utilities Functions

__iew_get_reason_code()
__iew_get_return_code()

__iew_putD() – Put data

Add data to, or replace data within, a specified class and section.

Format

```
#include <__iew_api.h>
int __iew_putD(_IEWAPIContext *__context,
                           const char *__class, const char *__section,
                           void **__data_entry,
                           int __count, int __cursor,
                           _IEWAPIFlags__flags);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__oldname
class name .

__section
section name.

__data_entry
input buffer, can be null when count is zero and __enddata is specified.

__count
number of data bytes or records to be inserted.

__cursor
starting position.

__flags
API flags:

- __enddata - whether all of the sections being added by the series of this calls are complete
- __newsect - whether a new section is being added to the workmod

Returned Value

If successful, __iew_putD() returns 0.

If unsuccessful, __iew_putD() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_get_return_code().

Utilities Functions

__iew_get_reason_code()
__iew_get_return_code()

__iew_rename() – Rename symbolic references

Obsolescent: intended for compatibility with programs processed by the prelinker. Can be used only when _IEW_BIND intent was specified.

Format

```
#include <__iew_api.h>
int __iew_rename(_IEWAPIContext *__context,
                            const char *__oldname,
                            const char *__newname);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__oldname
symbol to be renamed.

__newname
symbol to which the old name should be changed.

Returned Value

If successful, __iew_rename() returns 0.

If unsuccessful, __iew_rename() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_get_return_code().

Utilities Functions

__iew_get_reason_code()
__iew_get_return_code()

__iew_resetW() – Reset workmod

Clears all data from the active workmod to allow a different program to be brought in. This is required between any two include requests if _IEW_ACCESS was specified on the most recent __iew_openW() or __iew_resetW() call.

Format

```
#include <__iew_api.h>
int __iew_resetW(_IEWAPIContext *__context,
                              _IEWIntent __intent,
                              _IEWFlags __flags);
```

Parameters Descriptions

__context

API context is created and returned by calling __iew_openW() and is used throughout the open session.

__intent

processing intent can be one of the following:

- _IEW_ACCESS
- _IEW_BIND

Note: _IEW_ACCESS is more efficient but only allows the program to be inspected or copied.

__flags

API flags:

- __protect - whether to allow the binder to reset a workmod that has been altered but not yet saved or loaded.

Note: Set the flag to 1 to turn the bit on or 0 to turn the bit off.

Returned Value

If successful, __iew_resetW() returns 0.

If unsuccessful, __iew_resetW() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_get_return_code().

Utilities Functions

__iew_get_reason_code()
__iew_get_return_code()

__iew_saveW() – Save workmod

Stores the active workmod on DASD. Can be used either to store a new or modified program after __iew_bind() or to copy a module.

Format

```
#include <__iew_api.h>
int __iew_saveW(_IEWAPIContext *__context,
                              const char *__dd_or_path,
                              const char *_sname,
                              _IEWAPIFlags __flags);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__dd_or_path
DD name or pathname.

__sname
member name of the program to be saved in the target library.

__flags
API flags:

- __replace - whether or not the program module will replace an existing member of the same name in the target library.

Note: Set the flag to 1 to turn the bit on or 0 to turn the bit off.

Returned Value

If successful, __iew_saveW() returns 0.

If unsuccessful, __iew_saveW() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_get_return_code().

Utilities Functions

__iew_get_reason_code()
__iew_get_return_code()

__iew_setL() – Set library

Adds to or limit automatic library call processing. This services is equivalent to the LIBRARY control statement. Can be used only when _IEW_BIND intent was specifed.

Format

```
#include <__iew_api.h>
int __iew_setL(_IEWAPIContext *__context,
                        const char *__symbol,
                        _IEWLibOpttype __libopttype,
                        const char *__dd_or_path);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__symbol
symbol name.

__libopttype
automatic library call option can be one of the following:

- _IEW_CALL
- _IEW_NOCALL
- _IEW_EXCLUDE

Returned Value
If successful, __iew_setL() returns 0.

If unsuccessful, __iew_setL() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_get_return_code().

Utilities Functions
__iew_get_reason_code()
__iew_get_return_code()

__iew_setO() – Set option
Set binder options to be used for subsequent processing. See Chapter 7, “Setting options with the regular binder API,” on page 173 for details. Environment options cannot be specified here.

Format
```
#include <__iew_api.h>
int __iew_setO(_IEWAPIContext *__context,
                          const char *__parms);
```

Parameters Descriptions
__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__parms
list of binder options.

Returned Value
If successful, __iew_setO() returns 0.

If unsuccessful, __iew_setO() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_get_return_code().

Utilities Functions
__iew_get_reason_code()
__iew_get_return_code()

__iew_startS() – Start segment
Structures an overlay format program. Used in conjunction with __iew_orderS(). Can be used only when _IEW_BIND intent is specified.

Format
```
#include <__iew_api.h>
int __iew_startS(_IEWAPIContext *__context,
                           const char *__origin,
                           _IEWAPIFlags __flags);
```

Parameters Descriptions
__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__origin
segment origin symbol.

__flags
API flags:

- __region - whether or not the segment begins a new region within the program module

Note: Set the flag to 1 to turn the bit on or 0 to turn the bit off.

Returned Value

If successful, __iew_startS() returns 0.

If unsuccessful, __iew_startS() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_get_return_code().

Utilities Functions

__iew_get_reason_code()
__iew_get_return_code()

Binder API utility functions

This topic describes the binder API utility functions.

__iew_create_list() – Create list

The binder's lists consist of a fullword count of the number entries followed by the entries. Each list entry contains an 8-byte name, a fullword containing the length of the value string, and a 31-bit pointer to the value string. The __iew_create_list() function creates a list of keywords and values in "binder list format". It is used to support file lists and exit lists for the __iew_openW() call. There are two types of lists:

1. A list of DD names with keywords being any of the standard binder DD names as strings, and values being string replacement names to use for them.
2. A list of exit routines with keywords being any of the strings "MESSAGE", "INTFVAL", or "SAVE", and values being an array of three pointers, to:
   - exit routine entry point
   - application data passes to the exit
   - message exit severity (unused, but must be provided as zero, for the other two exits)

Format

```
#include <__iew_api.h>
_IEWList *__iew_create_list(int__size,
                                                    char *__keys[],
                                                    void *__values[]);
```

Parameters Descriptions

__size input: size of the list.

__keys
input: list of keywords.

__values
input: list of values.

Returned Value

If successful, __iew_create_list() returns API list.

If unsuccessful, __iew_create_list() returns null.

__iew_eod() – Test for end of data

The __iew_eod() function tests whether end of data condition has been set for a series of __iew_getC(), __iew_getD(), __iew_getE() or __iew_getN() calls.

Format

```
#include <__iew_api.h>
int __iew_eod(_IEWAPIContext *__context,
                          void **__data_entry,
                          const char *__class);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__data_entry
returned buffer – based on class name.

__class
class name (binder-defined class).

Note: Testing end of data condition for __iew_getN() call, specify “B_BNL” as class name.

Returned Value

If successful, __iew_eod() returns 0.

If unsuccessful, __iew_eod() returns nonzero.

__iew_get_reason_code() – Get a reason code from API context

The reason code identifies the nature of the problem. It is zero if the return code was zero, a valid reason code otherwise. Reason codes are in the format 0x83*eegggg*. *ee* is 00 except for logic errors and abends when it is EE or FF respectively. *gggg* contains an information code.

The __iew_get_reason_code() function returns binder API reason code which is stored in the API context from previous API call.

Format

```
#include <__iew_api.h>
unsigned int __iew_get_reason_code(_IEWAPIContext *__context);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

Returned Value

__iew_get_reason_code() returns binder API reason code.

__iew_get_return_code() – Get a return code from API context

A return code, indicating the completion status of the requested function, is returned by most of the API calls. Return codes are interpreted as follows:

- **0** - Successful completion of the operation.
- **4** - Successful completion, but an unusual condition existed.
- 8 - Error condition detected.
- **12** - Severe error encountered.
- **16** - Termination error.

The __iew_get_return_code() function returns binder API return code which is stored in the API context from previous API call. It is used as an alternative to retrieving the return code from API context.

Format

```
#include <__iew_api.h>
unsigned int __iew_get_return_code(_IEWAPIContext *__context);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

Returned Value

__iew_get_return_code() returns binder API return code.

__iew_get_cursor() – Get cursor value

A cursor contains the item number at which binder should begin processing. It is also updated at the end of a retrieval request so that it acts as a placeholder when all the selected data does not fit in the buffer provided. The cursor value is initially set to zero to tell the binder to begin with the first item.

The __iew_get_cursor() function returns the cursor value that is stored in a private data associated with the returned buffer from the previous API call.

Format

```
#include <__iew_api.h>
int __iew_get_cursor(_IEWAPIContext *__context,
                                 void **__data_entry,
                                 const char *__class,
                                 int *__cursor);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__data_entry
returned buffer – based on class name.

__class
class name (binder-defined class).

__cursor
starting position

Returned Value

If successful, __iew_get_cursor() returns 0.

If unsuccessful, __iew_get_cursor() returns nonzero.

__iew_set_cursor() – Set cursor value

A cursor contains the item number at which binder should begin processing. It is also updated at the end of a retrieval request so that it acts as a placeholder when all the selected data does not fit in the buffer provided. The cursor value is initially set to zero to tell the binder to begin with the first item.

The __iew_set_cursor() function sets the cursor value which is stored in a private data associated with the data buffer.

Format

```
#include <__iew_api.h>
int __iew_set_cursor(_IEWAPIContext *__context,
                                    void **__data_entry,
                                    const char *__class,
                                    int __cursor);
```

Parameters Descriptions

__context
API context is created and returned by calling __iew_openW() and is used throughout the open session.

__data_entry
data buffer – based on class name.

__class
class name (binder-defined class).

__cursor
starting position

Returned Value

If successful, __iew_set_cursor() returns 0.

If unsuccessful, __iew_set_cursor() returns nonzero.

Fast data functions

This topic describes the fast data functions.

__iew_fd_end() – End a session

The __iew_fd_end() deletes all the parameters in the context(_IEWFDContext), releases IEWBFDAT from memory, and deletes the context.

This is a C function equivalent to fast data request code EN.

Format

```
#include <__iew_api.h>
_IEWFDContext  *__iew_fd_end(_IEWFDContext *__context,
                                                    unsigned int *__return_code,
                                                    unsigned int *__reason_code);
```

Parameters Descriptions

__context
FD context is created and returned by calling __iew_fd_open() and is used throughout the open session.

__return_code
return code is passed back from request code EN.

__reason_code
reason code is passed back from request code EN.

Returned Value

If successful, __iew_fd_end() returns NULL.

If unsuccessful, __iew_fd_end() returns FD context.

Note: If an invalid context is passed, __iew_fd_end() returns NULL.

__iew_fd_getC() – Get compile unit list

Obtains source information about the program and the compile units form which it was constructed. The program object source "header record" returned by fast data in the CUI buffer for a program object in a PDSE will never identify the data set containing the object.

Format

```
#include <__iew_api.h>
int __iew_fd_getC(_IEWFDContext *__context,
                              int __culist[],
                              _IEWCUIEntry ** __cui_entry);
```

Parameters Descriptions

__context
FD context is created and returned by calling __iew_fd_open() and is used throughout the open session.

__culist
compile unit list – array of compile units where the first entry is used to specify the total number of compile unit entries. If the first entry is non zero, then one record for each compile unit in a list of compile units will be returned. If the first entry is zero, then one record of each of all compile units will be returned.

__cui_entry
returned buffer – CUI entry, one record for each compile unit in a list of compile units will be returned.

Returned Value

If successful, __iew_fd_getC() returns > 0 (count, number of entries returned in the buffer).

If unsuccessful, __iew_fd_getC() returns 0.

Utilities Functions

__iew_fd_eod()
__iew_fd_get_reason_code()
__iew_fd_get_return_code()
__iew_fd_get_cursor()
__iew_fd_set_cursor()

__iew_fd_getD() – Get data

Return data associated with a specified class (and optionally section) in the program.

Format

```
#include <__iew_api.h>
int __iew_fd_getD(_IEWFDContext *__context,
                              const char *__class, const char * __sect,
                              long long *__reloc,
                              void ** __data_entry);
```

Parameters Descriptions

__context
FD context is created and returned by calling __iew_fd_open() and is used throughout the open session.

__class
class name.

__sect
section name.

__reloc
relocation address.

__data_entry
returned buffer – based on class.

Returned Value

If successful, __iew_fd_getD() returns > 0 (count, could be number of bytes of TEXT (if class=TEXT) or number of entries returned in the buffer).

If unsuccessful, __iew_fd_getD() returns 0.

Utilities Functions

__iew_api_name_to_str()
__iew_fd_eod()
__iew_fd_get_reason_code()
__iew_fd_get_return_code()
__iew_fd_get_cursor()
__iew_fd_set_cursor()

__iew_fd_getE() – Get ESD data

Returns external symbol dictionary information selected by class and/or section to which it refers.

Format

```
#include <__iew_api.h>
int __iew_fd_getE(_IEWFDContext *__context,
                             const char *__sect, const char *__class,
                             _IEWESDEntry ** __esd_entry);
```

Parameters Descriptions

__context
FD context is created and returned by calling __iew_fd_open() and is used throughout the open session.

__sect
section name.

__class
class name.

__esd_entry
returned buffer – ESD entry.

Returned Value

If successful, __iew_fd_getE() returns > 0 (count, number of entries returned in the buffer).

If unsuccessful, __iew_fd_getE() returns 0.

Utilities Functions

__iew_api_name_to_str()
__iew_fd_eod()
__iew_fd_get_reason_code()
__iew_fd_get_return_code()
__iew_fd_get_cursor()
__iew_fd_set_cursor()

__iew_fd_getN() – Get names

Returns a list of section or class names within the program together with information about the sections or classes.

Format

```
#include <__iew_api.h>
int __iew_fd_getN(_IEWFDContext *__context,
                              _IEWNameType __nametype,
                              _IEWNameListEntry ** __name_entry);
```

Parameters Descriptions

__context
FD context is created and returned by calling __iew_fd_open() and is used throughout the open session.

__nametype
name type can be _IEW_SECTION or _IEW_CLASS

__name_entry
returned buffer – binder name list.

Returned Value

If successful, __iew_fd_getN() returns > 0 (count, number of entries returned in the buffer).

If unsuccessful, __iew_fd_getN() returns 0.

Utilities Functions

__iew_fd_eod()
__iew_fd_get_reason_code()
__iew_fd_get_return_code()
__iew_fd_get_cursor()
__iew_fd_set_cursor()

__iew_fd_open() – Open a session

The __iew_fd_open() will create a context (_IEWFDContext) and initialize all the parameters in the context and load IEWBFDAT into memory. If successful, the context is created and returned to user for all subsequent API calls.

Format

```
#include <__iew_api.h>
_IEWFDContext *__iew_fd_open(_IEWTargetRelease __release,
                                          unsigned int *__return_code,
                                          unsigned int *__reason_code);
```

Parameters Descriptions

__release

Target release can be one of the following:

- _IEW_ZOSV1R9
- _IEW_ZOSV1R10
- _IEW_ZOSV1R11
- _IEW_ZOSV1R12
- _IEW_ZOSV1R13

Note: It is recommended that you define the feature test macro _IEW_TARGET_RELEASE prior to the #include <__iew_api.h>, and that _IEW_TARGET_RELEASE be used for __release. This definition ensures that structure mappings in <__iew_api.h> are consistent with the data returned by the other API access functions.

__return_code

The return code.

__reason_code

The reason code.

Returned Value

If successful, __iew_fd_open() returns FD context.

If unsuccessful, __iew_fd_open() returns null.

__iew_fd_startDcb() – Starting a session with BLDL data

Begin inspecting a program object identified by a DCB and BLDL list entry.

Format

```
#include <__iew_api.h>
int __iew_fd_startDcb(_IEWFDContext *__context,
                                       void *__dcbptr,
                                       void *__deptr);
```

Parameters Descriptions

__context

FD context is created and returned by calling __iew_fd_open() and is used throughout the open session.

__dcptr

points to DCB.

__deptr

member entry returned by BLDL, at least 62 bytes long.

Returned Value

If successful, __iew_fd_startDcb() returns 0.

If unsuccessful, __iew_fd_startDcb() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_fd_get_return_code().

Utilities Functions

__iew_fd_get_reason_code()
__iew_fd_get_return_code()

__iew_fd_startDcbS() – Starting a session with a System DCB

A specialized version of __iew_fd_startDcb() used for a system DCB.

Format

```
#include <__iew_api.h>
int __iew_fd_startDcbS(_IEWFDContext *__context,
                                        void *__dcbptr,
                                        void *__deptr);
```

Parameters Descriptions

__context
FD context is created and returned by calling __iew_fd_open() and is used throughout the open session.

__dcptr
points to DCB.

__deptr
member entry returned by BLDL, at least 62 bytes long.

Returned Value

If successful, __iew_fd_startDcbS() returns 0.

If unsuccessful, __iew_fd_startDcbS() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_fd_get_return_code().

Utilities Functions

__iew_fd_get_reason_code()
__iew_fd_get_return_code()

__iew_fd_startName() – Starting a session with a DD name or path

Begin inspecting a program object identified either by a path name or by a DDname and member name.

Format

```
#include <__iew_api.h>
int __iew_fd_startName(_IEWFDContext *__context,
                                        const char *__dd_or_path,
                                        const char *__member);
```

Parameters Descriptions

__context
FD context is created and returned by calling __iew_fd_open() and is used throughout the open session.

__dd_or_path
DD name or path name.

__member
member name.

Returned Value

If successful, __iew_fd_startName() returns 0.

If unsuccessful, __iew_fd_startName() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_fd_get_return_code().

Utilities Functions

__iew_fd_get_reason_code()
__iew_fd_get_return_code()

__iew_fd_startToken() – Starting a session with a CSVQUERY token

Begin inspecting a program object identified by an entry point token returned by CSVQUERY.

Format

```
#include <__iew_api.h>
int __iew_fd_startToken(_IEWFDContext *__context,
                                        void *__eptoken);
```

Parameters Descriptions

__context
FD context is created and returned by calling __iew_fd_open() and is used throughout the open session.

__eptoken
point to EPTOKEN, token returned from the MVS CSVQUERY service.

Returned Value

If successful, __iew_fd_startToken() returns 0.

If unsuccessful, __iew_fd_startToken() returns nonzero.

Note: The returned value is the same as the code returned by a subsequent __iew_fd_get_return_code().

Utilities Functions

__iew_fd_get_reason_code()
__iew_fd_get_return_code()

Fast data utility functions

This topic describes the fast data utility functions.

__iew_fd_eod() – Test for end of data

The __iew_fd_eod() function tests whether end of data condition has been set for a series of __iew_fd_getC(), __iew_fd_getD(), __iew_fd_getE(), or __iew_fd_getN() calls.

Format

```
#include <__iew_api.h>
int __iew_fd_eod(_IEWFDContext *__context,
                            void **__data_entry,
                            const char *__class);
```

Parameters Descriptions

__context
FD context is created and returned by calling __iew_fd_open() and is used throughout the open session.

__data_entry
returned buffer – based on class name.

__class
class name (binder-defined class).

Note: Testing end of data condition after __iew_fd_getN() call, specify "B_BNL" as class name.

Returned Value

If successful, __iew_fd_eod() returns 0.

If unsuccessful, __iew_fd_eod() returns nonzero.

__iew_fd_get_reason_code() – Get a reason code from FD context

The reason code identifies the nature of the problem. It is zero if the return code was zero, a valid reason code otherwise. Reason codes are in the format 0x1080*gggg*. *gggg* contains an information code.

The __iew_fd_get_reason_code() function returns fast data reason code which is stored in the FD context from previous API call.

Format

```
#include <__iew_api.h>
unsigned int __iew_fd_get_reason_code(_IEWFDContext *__context);
```

Parameters Descriptions

__context
FD context is created and returned by calling __iew_fd_open() and is used throughout the open session.

Returned Value

__iew_fd_get_reason_code() returns fast data access reason code.

__iew_fd_get_return_code() – Get a return code from FD context

A return code, indicating the completion status of the requested function, is returned by most of the API calls. Return codes are interpreted as follows:

- **0** - Successful completion of the operation.
- **4** - Successful completion, but an unusual condition existed.

- **8** - Error condition detected.
- **12** - Severe error encountered.
- **16** - Termination error.

The __iew_fd_get_return_code() function returns API return code which is stored in the FD context from previous API call. It is used as an alternative way to retrieve return code from FD context.

Format

```
#include <__iew_api.h>
unsigned int __iew_fd_get_return_code(_IEWFDContext *__context);
```

Parameters Descriptions

__context
FD context is created and returned by calling __iew_fd_open() and is used throughout the open session.

Returned Value

__iew_fd_get_return_code() returns fast data access return code.

__iew_fd_get_cursor() – Get cursor value

A cursor contains the item number from which fast data should begin processing. It is also updated at the end of a retrieval request so that it acts as a placeholder when all the selected data does not fit in the buffer provided. The cursor value is initially set to zero to tell the fast data to begin with the first item.

The __iew_fd_get_cursor() function returns the cursor value that is stored in a private data associated with the returned buffer from the previous API call.

Format

```
#include <__iew_api.h>
int __iew_fd_get_cursor(_IEWFDContext *__context,
                                        void **__data_entry,
                                        const char *__class,
                                        int *__cursor);
```

Parameters Descriptions

__context
FD context is created and returned by calling __iew_fd_open() and is used throughout the open session.

__data_entry
returned buffer – based on class name.

__class
class name (binder-defined class).

__cursor
starting position.

Returned Value

If successful, __iew_fd_get_cursor() returns 0.

If unsuccessful, __iew_fd_get_cursor() returns nonzero.

__iew_fd_set_cursor() – Set cursor value

A cursor contains item number at which fast Ddata should begin processing. It is also updated at the end of a retrieval request so that it acts as a placeholder when all the selected data does not fit in the buffer provided. The cursor value is initially set to zero to tell the fast data to begin with the first item.

The __iew_fd_set_cursor() function sets the cursor value which is stored in a private data associated with the data buffer.

Format

```
#include <__iew_api.h>
int __iew_fd_set_cursor(_IEWFDContext *__context,
                                        void **__data_entry,
                                        const char *__class,
                                        int __cursor);
```

Parameters Descriptions

__context
FD context is created and returned by calling __iew_fd_open() and is used throughout the open session.

__data_entry
returned buffer – based on class name.

__class
class name (binder-defined class).

__cursor
starting position.

Returned Value

If successful, __iew_fd_set_cursor() returns 0.

If unsuccessful, __iew_fd_set_cursor() returns nonzero.

Binder API and fast data API common utility functions

__iew_api_name_to_str() – Convert API name into string

Data buffers returned by the binder and fast data APIs often contain pointers to names. Those are character data, but not stored as C null-delimited strings. Instead they have a separate length field in the buffer structure. This function returns a C string equivalent to the name returned by the API.

The function also provides special handling for binder-generated names, which are returned as binary numbers. The function will convert those to special displayable strings which will be recognized and automatically reconverted by the functions in this suite if they are passed back to the API later.

Format

```
#include <__iew_api.h>
void __iew_api_name_to_str(const char *__name,
                                                short __len,
                                                char *__str);
```

Parameters Descriptions

__name
input: varying string characters.

__len input: varying string size.

__str output: string.

Note: You need to allocate storage for string. The length should be the greater of 10 and input len+1.

Returned Value

None.

Binder API return and reason codes

The following binder API return and reason codes are added to support binder C/C++ API. These are in addition to the return and reason codes documented at the end of “IEWBIND function reference” on page 24.

Return Code (decimal)	Reason Code (hexadecimal)	Explanation
16	83001001	Allocation error, not enough space to satisfy the request for space allocation.
16	83001002	Fetch() failed, IEWBIND module could not be loaded into memory.
12	83001003	Invalid API context, API context was either NULL or invalid.
12	83001004	Invalid release, the target release level is not supported.
12	83001005	Unsupported release, the target release level is not supported.
12	83001006	Incorrect buffer entry, the buffer entry and the class name need to match with previous call.

Fast data access return and reason codes

The following fast data access return and reason codes are added to support binder C/C++ API. These are in addition to the return and reason codes documented at the end of Chapter 4, “IEWBFDAT - Binder Fast data access API functions,” on page 109.

Return Code (decimal)	Reason Code (hexadecimal)	Explanation
16	10801001	Allocation error, not enough space to satisfy the request for space allocation.
16	10801002	Fetch() failed, IEWBFDAT module could not be loaded into memory.
12	10801003	Invalid FD context, FD context was either NULL or invalid.
12	10801004	Invalid release, the target release level is not supported.
12	10801005	Unsupported release, the target release level is not supported.
12	10801006	Incorrect buffer entry, the buffer entry and the class name need to match with previous call.

Chapter 6. Invoking the binder program from another program

Programming Interface information

You can pass control to the binder from a program in one of two ways:

1. As a subprogram, with the execution of a CALL macro instruction (after the execution of a LOAD macro instruction), a LINK macro instruction, or an XCTL macro instruction.
2. As a subtask with the execution of the ATTACH macro instruction.

24-bit or 31-bit addressing can be used with any of these macros.

The syntax of the macros used to invoke the binder follows:

[*symbol*]	**{ATTACH|**	**EP=***bindername,*
	LINK|	**PARAM=(***optionlist***[***,ddname list***]),**
	XCTL}	**VL=1**
[*symbol*]	**LOAD**	**EP=***bindername*

EP=*bindername*

Specifies a symbolic name of the binder. You use these names to invoke the binder for the indicated services:

IEWBLINK Binds a program module and stores it in a program library. Alternative alias names are IEWL, LINKEDIT, HEWL, and HEWLH096.

IEWBLOAD Binds a program module and loads it into virtual storage, but does not identify it. Upon return from IEWBLOAD,

- Register 0 contains the entry point address of the loaded program. The high order bit is set for AMODE(31) or AMODE(ANY). The low order bit is set for AMODE(64). In addition, the top 32 bits of extended Register 0 are cleared for AMODE(64).
- Register 1 contains the address of a two-word area containing the following information:
  - Word 1 contains the address of the beginning of the virtual storage occupied by the loaded program.
  - Word 2 contains the size in bytes of the virtual storage occupied by the loaded program.

Alternative alias names are IEWLOADR, HEWLOADR, and IEWLOAD.

IEWBLODI Binds a program module, loads it into virtual storage, and identifies it to the system using the IDENTIFY macro. Upon return from IEWBLODI,

- Register 0 contains the entry point address of the loaded program. The high order bit is set for AMODE(31) or AMODE(ANY). The low order bit is set for AMODE(64). In addition, the top 32 bits of extended Register 0 are cleared for AMODE(64).
- Register 1 contains the address of an 8 byte field containing the module name used on the IDENTIFY macro.

© Copyright IBM Corp. 1991, 2012

Alternative alias names are IEWLOADI and HEWLOAD.

IEWBLDGO Binds a program module, loads it into virtual storage, and executes it. Alternative alias names are IEWLDRGO, LOADER, and HEWLDRGO.

PARAM=(*optionlist***[,***ddname list***])**
Specifies, as a sublist, address parameters to be passed from the program to the binder.

optionlist
Specifies the address of a variable-length list containing the options and attributes. This address must be provided even if no list is provided.

The option list must begin on a half-word boundary. The two high-order bytes contain a count of the number of bytes in the remainder of the list. If no options or attributes are specified, the count must be zero. The option list is free form, with each field separated by a comma. No blanks or zeros should appear in the list. See *z/OS MVS Program Management: User's Guide and Reference* for more information about processing and attribute options.

ddname list
Specifies the address of a variable-length list containing alternative ddnames for the data sets used during binder processing. If standard DD names are used, this operand can be omitted.

The DD name list must begin on a halfword boundary. The 2 high-order bytes contain a count of the number of bytes in the remainder of the list. Each name of less than 8 bytes must be left justified and padded with blanks. If an alternate ddname is omitted from the list, the standard name will be assumed. If the name is omitted within the list, the 8-byte entry must contain binary zeros. Names can be omitted from the end by merely shortening the list.

The sequence of the 8-byte entries in the ddname list is as follows:

Entry	Alternate Name For:
1	SYSLIN
2	Member name (the name under which the output module is stored in the SYSLMOD data set; this entry is used if the name is not specified on the SYSLMOD DD statement or if there is no NAME control statement)
3	SYSLMOD
4	SYSLIB
5	Not applicable
6	SYSPRINT or SYSLOUT
7-11	Not applicable
12	SYSTERM
13	SYSDEFSD

VL=1
Specifies that the sign bit is set to 1 in the last fullword of the address parameter list.

When the binder completes processing, a return code is returned in register 15. See "Binder return codes" in *z/OS MVS Program Management: User's Guide and Reference* for a list of return codes.

For more information on the use of these macro instructions and for the syntax of the CALL macro, see *z/OS MVS Programming: Assembler Services Guide*.

You must code optionlist and VL1 on CALL.

End of Programming Interface information

Chapter 7. Setting options with the regular binder API

Options can be specified in a number of different ways. Besides those defined in this topic, you can find other ways to specify options in *z/OS MVS Program Management: User's Guide and Reference*.

Table 47 provides a list of options that can be specified by the following sources, all of which are defined in this manual:

- Parameter 1 on ATTACH, LINK, XCTL, or CALL as defined in Chapter 6, "Invoking the binder program from another program," on page 169
- The PARMS operand of either SETO ("SETO: Set option" on page 88) or STARTD ("STARTD: Start dialog" on page 91)
- The OPTION and OPTVAL operands of SETO ("SETO: Set option" on page 88)
- The OPTIONS operand of STARTD ("STARTD: Start dialog" on page 91)
- The ability to pass options on the IEWBIND_OPTIONS envar when STARTD ENVARS= is used ("STARTD: Start dialog" on page 91)

Setting options with the binder API

Many options can be set by using STARTD and SETO to specify options and values. You can set most of the options that can be set in the PARM field of the EXEC statement and there are a few options for use only through the API functions.

Any option can be set using STARTD call. All options except the CALLERID, COMPAT, EXITS, LINECT, MSGLEVEL, OPTIONS, PRINT, TERM, TRAP, and WKSPACE options can also be set using the SETO call (or the SETOPT control statement). Negative keywords (for example, NOLIST or NOMAP) cannot be used with the OPTION operand of SETO or the options list of the STARTD. These options are set by assigning the value NO to the primary option. Refer to the descriptions of the STARTD and SETO operands for further information.

Note: Environmental options including CALLERID, COMPAT, LINECT, MSGLEVEL, PRINT, TERM, TRAP and WKSPACE are not valid for SETO.

Table 47 lists options with allowable and default values. Numeric values are coded as numeric character strings unless otherwise specified. The table also references any corresponding batch options.

Table 47. Setting options with the binder API

Option	Description	Allowable Values	Default Value
AC	Sets the APF authorization code in the saved module. Also see AC option in *z/OS MVS Program Management: User's Guide and Reference*.	0 - 255	0
ALIASES	Allows you to mark external symbols as aliases for symbol-resolution purposes.	ALL/NO	NO
ALIGN2	Requests 2KB page alignment. Also see ALIGN2 option in *z/OS MVS Program Management: User's Guide and Reference*.	Y, N	N
AMODE	Sets the addressing mode. For a detailed description, see AMODE option in *z/OS MVS Program Management: User's Guide and Reference*.	24, 31, 64, ANY, MIN	Derived in ESD
CALL/NCAL	Allows or disallows automatic library call. Also see CALL option in *z/OS MVS Program Management: User's Guide and Reference*.	Y, N	Y
CALLIB	Specifies the library for automatic call.	ddname of 1 to 8 characters	None

© Copyright IBM Corp. 1991, 2012

Table 47. Setting options with the binder API (continued)

Option	Description	Allowable Values	Default Value
CALLERID	Specifies string to be printed at the top of each page of the binder output listings.	Character string of up to 80 bytes	None
CASE	Specifies case sensitivity for symbols. Also see CASE option in *z/OS MVS Program Management: User's Guide and Reference*.	UPPER, MIXED	UPPER
COMPAT	Specifies binder compatibility level.	LKED, PM1, PM2, PM3, PM4, CURRENT, MIN, Release identifier (for example: ZOSV1R7)	MIN
COMPRESS	Controls whether the binder should attempt to compress non-program data to reduce DASD space.	NO, AUTO, YES	AUTO
DC	Allows compatibility with down-level software. Also see DC option in *z/OS MVS Program Management: User's Guide and Reference*.	Y, N	N
DCBS	Allows the block size of the SYSLMOD data set to be reset for PDS only. Also see DCBS option in *z/OS MVS Program Management: User's Guide and Reference*.	Y, N	N
DYNAM	Determines whether a module being bound is enabled for dynamic linking.	DLL, NO	NO
EDIT	Requests external symbol data to be retained, allowing later reprocessing. Also see EDIT option in *z/OS MVS Program Management: User's Guide and Reference*.	Y, N	Y
EP	Specifies the program entry point (name,[offset]). Also see EP option in *z/OS MVS Program Management: User's Guide and Reference*.	(symbol of up to 32767characters [,integer value])	None
EXITS	Specifies user exits.	INTFVAL	None
EXTATTR	Specifies extended attributes for SYSLMOD when saved in a z/OS UNIX file.	See EXTATTR option in *z/OS MVS Program Management: User's Guide and Reference*.	system setting for SYSLMOD file
FETCHOPT	Sets loading options for program objects. Also see FETCHOPT option in *z/OS MVS Program Management: User's Guide and Reference*. PACK and PRIME are ignored	PACK or NOPACK, PRIME or NOPRIME	NOPACK, NOPRIME
FILL	Specifies that uninitialized areas of module are to be filled with the byte provided.	Any byte value	None
GID	Specifies the Group ID attribute to be set for the SYSLMOD file.	A string of up to 8 alphanumeric characters which represents a RACF® group name or a numeric z/OS UNIX group ID	system setting for SYSLMOD file
HOBSET	Instructs the binder to set the high-order bit in V-type adcons according to the AMODE of the target.	YES | NO	NO
IEWPARMS	Provides an alternate source for binder options. See *z/OS MVS Program Management: User's Guide and Reference* for more information.	80-byte	None
INFO	See *z/OS MVS Program Management: User's Guide and Reference* for more information.	INFO | INFO=NO | NOINFO	INFO=NO
LET	Allows errors of a specified severity to be accepted. Also see LET option in *z/OS MVS Program Management: User's Guide and Reference*.	0, 4, 8, 12	4
LINECT	Specifies number of lines per page of the binder output listings. Also see LINECT option in *z/OS MVS Program Management: User's Guide and Reference*.	Integer value	60
LIST	Controls contents of the output listings. Also see LIST option in *z/OS MVS Program Management: User's Guide and Reference*.	OFF, SUMMARY, STMT, ALL NOIMPORT, NOIMP	SUMMARY
LNAME	Specifies the program name to be identified to the system. Also see LNAME parameter description in "LOADW: Load workmod" on page 69.	Symbol of up to 8 characters	

Table 47. Setting options with the binder API (continued)

Option	Description	Allowable Values	Default Value
MAP	Requests a module map. Also see MAP option in *z/OS MVS Program Management: User's Guide and Reference*.	Y, N	N
MAXBLK	Specifies the maximum record length for the text of a program module. Also see MAXBLK option in *z/OS MVS Program Management: User's Guide and Reference*.	256 - 32760	32760
MODLIB	Specifies a ddname for the output program library. Also see MODLIB parameter description in "SAVEW: Save workmod" on page 81.	ddname of up to 8 characters	None
MODMAP	See *z/OS MVS Program Management: User's Guide and Reference* for more information.	NO | LOAD | NOLOAD	NO
MSGLEVEL	Specifies the minimum severity level of messages to be issued. Also see MSGLEVEL option in *z/OS MVS Program Management: User's Guide and Reference*.	0, 4, 8, 12, 16	0
OL	Limits how the program can be brought into virtual storage. Also see OL option in *z/OS MVS Program Management: User's Guide and Reference*.	Y, N	N
OVLY	Requests the program be bound in overlay format. Also see OVLY option in *z/OS MVS Program Management: User's Guide and Reference*.	Y, N	N
PATHMODE	Specifies pathmode to be used when saving a module to an z/OS UNIX file.	See PATHMODE option in *z/OS MVS Program Management: User's Guide and Reference*.	Owner permission for read, write, and execute
PRINT	Requests that messages be written to the SYSLOUT or SYSPRINT data set. Also see PRINT option in *z/OS MVS Program Management: User's Guide and Reference*.	Y, N	Y
RES	Requests that the link pack area queue be searched to resolve references for programs that will not be saved. Also see RES option in *z/OS MVS Program Management: User's Guide and Reference*.	Y, N	N
REUS	Specifies the reusability characteristics of the program module. Also see REUS in *z/OS MVS Program Management: User's Guide and Reference*.	NONE, SERIAL, RENT, REF	NONE
RMODE	Sets the residence mode. Also see RMODE in *z/OS MVS Program Management: User's Guide and Reference*.	24, ANY, SPLIT	Derived from ESD
SCTR	Requests scatter format. Used only for MVS system nucleus load module.	Y, N	N
SIGN	Builds a module signature.	Y, N	N
SNAME	Specifies a member name for a saved program module. Also see SNAME parameter description in "SAVEW: Save workmod" on page 81.	Member name of 1 to 1024 characters	None
SSI	Specifies a system status index. Also see SSI option in *z/OS MVS Program Management: User's Guide and Reference*.	8 hexadecimal digits	None
STORENX	Prevents an executable module from being replaced by a non-executable module in the target library. Also see STORENX option in *z/OS MVS Program Management: User's Guide and Reference*.	Y, N	N
STRIPCL	Removes unneeded classes from a program object or load module. Also see STRIPCL option in *z/OS MVS Program Management: User's Guide and Reference*.	YES, NO	NO
STRIPSEC	Removes unreferenced unnamed sections from a program object or load module. Also see STRIPSEC option in *z/OS MVS Program Management: User's Guide and Reference*.	PRIV, YES, NO	NO
TERM	Requests that messages be sent to the terminal data set SYSTERM.. Also see TERM option in *z/OS MVS Program Management: User's Guide and Reference*.	Y, N	Y

Table 47. Setting options with the binder API (continued)

Option	Description	Allowable Values	Default Value
TEST	Requests that the program module be prepared for the TSO TEST command. Also see TEST option in *z/OS MVS Program Management: User's Guide and Reference*.	Y, N	N
TRAP	Controls error trapping. See *z/OS MVS Program Management: User's Guide and Reference* for more information.	OFF, ON, ABEND	ON
UID	Specifies the User ID attribute to be set for the SYSLMOD file.	A string of up to 8 alphanumeric characters which represents a user name (such as a TSO logon ID) or a numeric z/OS UNIX user ID	system setting for SYSLMOD file
UPCASE	While processing XOBJs, determines whether symbols should be upper-cased before finalizing symbol resolution during binding.	Y, N	N
WKSPACE	Specifies the amount of space available for binder processing, both below and above the 16 MB line (*value1*, *value2*). Also see WKSPACE option in *z/OS MVS Program Management: User's Guide and Reference*.	Values specified in units 96KB below 16MB and 1024KB above 16MB	All available space
XCAL	Requests that valid exclusive references between program segments of an overlay program module be allowed. Also see WKSPACE option in *z/OS MVS Program Management: User's Guide and Reference*.	Y, N	N
XREF	Requests printing of a cross-reference table. Also see XREF option in *z/OS MVS Program Management: User's Guide and Reference*.	Y, N	N

Chapter 8. User exits

User exits identify points in binder processing when the calling program regains control. The binder passes the user exit routine a data buffer. The exit routine can examine or modify the data, return it to the binder, and set a return code.

The binder provides user exit points for:
- All messages (or those above a specified severity) issued during processing
- Each successful or failed attempt to save a program object or load module
- Each linkage of an external reference to its target

You can define the user exits either using the EXITS parameter on a FUNC=STARTD call or by specifying them in the EXITS invocation option. Note that invocation options are passed one of four ways:
- Through the JCL PARM string for batch invocation
- Via PARAM when using ATTACH, LINK, or SCTL
- Pointed to by register 1 when using LOAD and CALL
- Within the OPTIONS string of FUNC=STARTD when using the API

If the EXITS option is found in the OPTIONS string passed by FUNC=STARTD when the API is used, that will override any EXITS parameter also passed by FUNC=STARTD.

Note: In this topic, you will see several references to "varying string." Keep in mind that a varying string consists of a halfword length field followed by a character string of that length and that varying strings are always addressed by pointing to the length field.

Execution environment

The execution environment of a user exit routine is:
- Enabled for I/O and external interrupts
- Holds no locks
- In task control block (TCB) mode
- With PSW key equal to the key of the issuer of IEWBIND
- In primary address space mode
- In 31-bit addressing mode
- In same PSW state as when IEWBIND was invoked
- With no FRRs on the current FRR stack.

Registers at entry to the user exit routine

A user exit routine is called using MVS linkage conventions:
- Register 1 contains the address of the parameter list.
- Register 13 contains the address of an 18-word register save area.
- Register 14 contains the return address to the binder.
- Register 15 contains the address of the exit routine's entry point.

© Copyright IBM Corp. 1991, 2012

Message exit

The binder passes control to your message user exit routine just prior to writing a message to the print data set (typically SYSPRINT). The message is stored in a buffer and passed to your routine. The exit routine can either prevent or allow the message to print.

When specifying this exit on STARTD, you provide the following information by means of the **EXITS** keyword:

- The entry point address of the message exit routine.
- Optional user data that is passed directly to the exit without being processed by the binder. This is typically used as an address to dynamic storage when the binder is invoked by a reentrant program, but can be a counter or other data.
- An address of a fullword containing an error severity level below which the binder does not call your exit routine. For example, if you code 8 in this word, the binder does not call your exit routine for normal output, informational messages, or warning messages.

The binder passes a parameter list to the exit routine. Register one contains the address of a list of addresses pointing to the following data:

- The user data specified in the second word of the exit specification.
- A pointer to a varying-length character buffer that contains a message. The length of the buffer is indicated in the first two bytes, which are not included in this length. The length can include trailing blanks.
- A halfword count of the number of lines in the buffer.
- A halfword count of the number of characters per line in the buffer.
- A 4-character message number extracted from the message text in the buffer.
- A halfword severity code.
- A fullword reason code to be set by the exit routine.
- A fullword return code to be set by the exit routine.

The exit routine can examine the messages but should not modify them. Before returning to the binder, your exit routine should set a return code for the binder to control its next action as follows:

Return Code	Explanation
00	Continue processing as if the exit routine had not been called.
04	Suppress printing of the message.

Save exit

The binder passes control to your save exit routine just prior to rejecting a primary member name or an alias name or after an attempt to save a primary member name or an alias name. The save operation might have succeeded or failed. The binder passes your routine information about the disposition of the name and, in certain situations, you can request that the save be retried.

Note: This exit is not invoked if the target is a z/OS UNIX System Services file.

When specifying this exit on STARTD, you provide the following information by means of the **EXITS** keyword:

- The entry point address of the exit routine.
- Optional user data that is passed directly to the exit without being processed by the binder. This is typically used as an address to dynamic storage when the binder is invoked by a reentrant program, but can be a counter or other data.
- A third fullword that currently is not used.

The binder passes a parameter list to the exit routine. Register one contains the address of a list of addresses to the following data:

- The user data specified in the second word of the exit specification.
- A fullword reason code (see below).
- A one-character EBCDIC argument identifying the type of name:

'P' for primary member name and **'A'** for alias name.

- A two-byte field specifying the length of the name, followed by the name
- A fullword return code to be set by the exit.

The reason codes passed to the exit by the binder are as follows:

Reason Code	Explanation
00000000	Normal completion.
83008000	No valid member name.
83008001	Duplicate member name.
83008010	Not-executable module replacing executable module.
83008050	Alias name is too long (PDS only).
83008051	Duplicate alias name.
83008052	No ESD for alias target.
83008060	Insufficient virtual storage for STOW (PDS only).
83008070	I/O error in directory.
83008071	Out of space in directory (PDS only).
83008078	Member or alias name not processed (PDSE only).
83008079	Miscellaneous error condition.

The save attempt can be retried only if the reason code is less than 83008050.

Your exit routine should examine the name and reason code. It can either retry the save, retry the save under a new name, or do nothing. Your routine should change the name in the parameter list if a save under a new name is requested.

Your exit routine must set a return code for the binder to control its next action as follows:

Return Code	Explanation
00	Continue processing as if the exit routine had not been called.
04	Retry the save with a new name. Your exit routine updates the fourth parameter in the parameter list with the new name. Valid for reason codes 83008000, 83008001, and 83008010.
08	Retry the original save and force a replacement. Valid for reason codes 83008001 and 83008010.

The binder performs the retry operation if requested. Your exit routine can be called repeatedly until the save is successful. There is no limit to the number of retry attempts. Your exit routine must return a zero return code eventually or a never-ending loop could occur.

Interface validation exit

The Interface Validation Exit (INTFVAL) allows your exit routine to examine descriptive data for both caller and called at each external reference. The exit can perform audits, such as examining parameter passing conventions, the number of parameters, data types, and environments. It can accept the interface, rename the reference, or leave the interface unresolved.

When specifying this exit on STARTD, you provide the following information by means of the **EXITS** keyword:

- The entry point address of the exit routine.
- Optional user data that is passed directly to the exit without being processed by the binder. This is typically used as an address to dynamic storage when the binder is invoked by a reentrant program, but can be a counter or other data.
- A third fullword that currently is not used.

The binder passes control to your exit routine at the completion of input processing, including autocall. This is before binding the module. If specified, the exit routine is invoked at three different points in binder processing, indicated by the function code passed to the exit:

'S' (Start) - Allows the exit routine to set up its environment and return a list of requested IDRs (product identifiers) to the binder.

'V' (Validate) - Allows the exit routine to validate all of the resolved and unresolved references from one section (CSECT) in the module to external labels in other sections.

'E' (End) - Invoked immediately prior to binding. Allows cleanup by the exit routine.

For each module being bound, Start will be called once, Validate will be called zero or more times, and End will be called once.

At the completion of autocall, the exit will be taken once for the Start function and once for each section in the module containing one or more *unchecked* external references. An external reference is unchecked if the *signature* in its ER record is either null (binary zeros) or does not match the signature in the LD.

An anchor word will be passed to the exit to provide for persistent storage between invocations. If the binder is invoked as a batch program, the anchor will be allocated and set to zero by the binder on invocation of the Start function; otherwise, the value passed to the binder on the STARTD exit specification will be passed through to the exit. If the exit routine provides its own dynamic storage, that address can be stored in the anchor word for addressability by subsequent invocations. No facilities will be available for trapping errors: If the exit fails, the binder fails. A message can be returned by the exit routine, however, to be printed by the binder.

The binder passes a parameter list to the exit routine. Register one contains the address of a list of addresses to the following data:

- Function Code ('S', 'V' or 'E'). A 1-byte function code indicating Start, Validate or End, respectively.
- Anchor Word. A 4-byte pointer variable for use by the exit.
- Exit control. The character string passed on the exit specification. The character string is immediately preceded by a halfword length field.
- Section name ('V' only). A varying string containing the name of the section being validated.
- Section vaddr ('V' only). A 4-byte pointer to the beginning of the first text element in the section being validated. This might not be useful in a multiclass module, since there is no designated *primary* class. This field is reserved for future use.
- Section IDRL. This parameter has a dual use, depending on the Function Code. On the Start ('S') call, the parameter is initialized to zero by the binder and optionally reset by the exit routine. On the Validate ('V') call, the binder sets the parameter to point to the IDR entry, if one is present, for the current section or to zero.
  - ('S') A 4-byte pointer, initialized to zero by the binder and optionally reset to the address of an IDR list by the exit routine. The list, if specified, indicates those product identifiers of interest to the exit. If the list is not returned by the exit, the exit will be invoked for all sections. See below for a discussion of the IDR selection list.
  - ('V') A 4-byte pointer, set by the binder, to the IDR entry for the section in process. The IDR data is preceded by a halfword length field.
- Reference List ('V' only). A 4-byte pointer to a list of unchecked references. See below for a discussion of the reference list.
- Return code. A fullword return code indicating the overall status of the exit. It will be initialized to zero on invocation of the exit.

  Return codes from the exit routine are:

  0 No further processing required of this section. The action code for all references is zero.

  4 Further processing required by the binder, as indicated in the returned action codes.

  12 Severe error. Make no more calls to the exit and do not save the module (unless LET=12).

  16 Terminate binder processing immediately
- Returned message. A 4-byte pointer to a varying string allocated by the exit and containing a message to be printed by the binder. The returned message must not be longer than 1000 bytes. The binder will prefix the returned message with its own message number.

  The message will be initialized to the null string so that the exit routine need not take any action unless a message is issued.

The IDR selection list is built by the exit routine on the “Start” call and its address returned to the binder. The purpose of the list is to improve performance by limiting the number of exit invocations. If a section to be validated was compiled by a language product not on the list, or if no IDR information is available for that section, the exit will not be taken. Each entry in the list consists of:

- A 4-byte address of another entry (zero indicates last entry)
- A 2-byte length containing the number of IDR bytes to be compared (1-14)

- A character string containing a substring of the product id, version and modification level. The string can contain a maximum of 14 bytes in the format ppppppppppvvmm.

The reference list is a linked list containing one entry for each unchecked ER in the section. References marked NOCALL or NEVERCALL will not be included in the list. The last entry in the list contains zero in the link field. A reference entry is 64 bytes in length and consists of the following fields:

```
OFF   TYPE      LEN   NAME          DESCRIPTION                   NOTES
---   --------  --- --------------  ----------------------------  -----
(0)   Address    4  REFL_NEXT       Address of next list entry      3
(4)   Address    4  REFL_T_SYMBOL   Address of referenced symbol    2
(8)   Address    4  REFL_T_SECTION  Address of target section name 2,8
(C)   Address    4  REFL_T_ELEMENT  Address of target element      1,8
(10)  Address    4  REFL_T_DESCR    Address of target descriptors  1,8
(14)  Address    4  REFL_T_IDR      Address of target IDR         1,5,8
(18)  Bit        4  REFL_T_ENVIR    Target environment            1,6,8
(1C)  Character  8  REFL_T_SIGN     Target signature                8
(24)  Address    4  REFL_T_ADCONS   Adcon list anchor               1
(28)  Address    4  REFL_C_DESCR    Address of caller descriptors   1
(2C)  Bit        4  REFL_C_ENVIR    Caller's environment           1,6
(30)  Character  8  REFL_X_SIGN     Exit signature                 4,9
(38)  Address    4  REFL_X_SYMBOL   New symbol (Char(*) varying)  4,7,9
(3C)  Unsigned   2  REFL_X_ACTION   Action code                    4,9
(3E)  Unsigned   2                  Reserved                        1

Notes:
  1   Must be zero.
  2   Points to varying character string.  String must begin with a
      halfword length field containing current length, excluding length
      field.
  3   Last entry in list set to zero.
  4   Output field.  Set by exit routine.
  5   IDR data is returned in the following format:
        0  CHAR    10  Processor Identification
       10  CHAR     2  Processor Version
       12  CHAR     2  Processor Modification Level
       14  CHAR     7  Date Compiled or Assembled (yyyyddd)
      The above 21-byte structure is preceded by a halfword length.
      The length can contain zero or any multiple of 21, allowing
      for multiple IDRs.
  6   Environmental bit settings are not yet defined.
  7   The exit routine must allocate and initialize a varying length
      character string, consisting of a halfword length field,
      containing the length of the symbol, immediately followed
      by the symbol itself.  The address of this varying string
      must be stored in the REFL_X_SYMBOL field in the reference
      list.
  8   Target fields will contain binary zeros for unresolved references
  9   Output fields will be initialized to binary zeros on invocation
      of the exit routine.
 10   Referenced symbol or section is in the following format:
              0   BIN    2    Length of name field.
              2   CHAR   *    Name.
```

Action codes set for each reference include the following:

0 No special processing required for this reference, such as changing the bind status flags, renaming the reference or storing signatures.

1 Validation successful. Store the exit signature in both LD and ER records.

2 Validation successful. Store glue code address in all referring adcons and store the exit signature in both LD and ER records.

3 Accept unresolved reference. Do not store the exit signature in the ER record. Reference will be treated as a weak reference and will not affect the return code from the binder.

4 Retry. New symbol has been provided for reference. Do not store signatures at this time. Reprocess autocall, if necessary, and revalidate.

5 Validation failed. Mark reference unresolved and do not store signatures at this time. The return code from the binder will reflect that there was at least one unresolved reference.

Post Processing:

On return from the exit, the binder will take the action described by the action code for each external reference. If the action code is 1 or 2, the signature returned by the exit will be stored, if nonzero, in both ER and LD, so that the interface will not be reexamined on a subsequent invocation. If the exit elects to rename a reference, the symbol will be changed FOR THAT REFERENCE ONLY. If the renamed reference cannot be resolved from labels already present in the module, another autocall pass will be required.

The exit routine should always return some kind of signature to the binder, if the interface is valid, so that the same reference does not get revalidated on subsequent passes. If the input signature is not null (binary zeros) return it to the binder; otherwise, return any 8-byte string (for example, date/time), which will then be saved in the LD and ER records. The signature can consist of any characters. The binder map, if requested, will indicate those sections that were included as a result of one or more renamed references. The flag position in the map, which normally contains an asterisk for autocalled sections, will be reset to “R”. There will be no indication as to the number and location of such renamed references nor their original names.

Default Exit Routine:

In the absence of an interface exit specification, the binder will default the exit processing. During default validation, the binder will examine each resolved ER - LD pair, checking the following:

- Compare the text type expected (ER) with that of the target (LD). Fifteen text types are supported by the binder, and can be passed in the ER and LD records via the binder API or the new object module (GOFF):

  ```
  0 - unspecified
  1 - data
  2 - instructions
  3-15 - for translator use
  ```

  If the text type for either caller or called is unspecified, or if the two text types are equal, the interface is considered valid.
- Compare the signature fields. If either is unspecified (binary zeroes) or if they match, the interface will be considered valid.

If either result is not valid, a warning message will be issued by the binder and the return code set to 4. The reference remains resolved, however.

Appendix A. Object module input conventions and record formats

This topic contains general-use programming interface and associated guidance information.

This topic contains binder input conventions and record formats for object modules.

See “Extended object module (XOBJ)” on page 191 for a format which may be generated by IBM C, C++, and PL/I compilers.

Appendix C, “Generalized object file format (GOFF),” on page 203 discusses support for the generalized object file format.

Input conventions

All object modules used as input to the binder must follow a number of input conventions. The binder treats violation of the following conventions as errors:

- All text records of a control section must follow the external symbol dictionary (ESD) record containing the control section (SD) or private code (PC) entry that describes the control section.
- The end of every object module must be marked by an END record.
- Each object module can contain only one zero-length control section (a control section whose length field in its SD or PC entry in the ESD contains zeros) if that control section has text records in the object module. The length must be specified on the END record of any module that contains a zero-length control section with text.
- Any relocation dictionary (RLD) item must be read after the ESD items to which it refers; if it refers to a label in a different control section, it must be read after the ESD item for that control section.
- The language translators must gather RLD items in groups of identical position pointers. No two RLD items having the same P pointer can be separated by an RLD item having a different P pointer.
- Each record of text[1] and each LD or LR entry in the ESD record must refer to an SD or PC entry in the ESD.
- The position pointer of every RLD item must point to an SD or PC entry in the ESD.
- No LD or LR can have the same name as an SD or CM.
- All SYM records must be placed at the beginning of an object module. The ESD for an object module containing test translator statements must follow the SYM records and precede TXT records.
- The binder accepts TXT records that are out of order within a control section, even though binder processing might be affected. TXT records are accepted even though they might overwrite previous text in the same control section. The binder does not eliminate any RLD items that correspond to overwritten text.
- During a single execution of the binder, if two or more control sections having the same name are read in, only the first control section is accepted; the subsequent control sections are deleted.

1. A common (CM) control section cannot contain text or external references.

© Copyright IBM Corp. 1991, 2012

- The binder interprets common (CM) entries in the ESD (blank or with the same name) as references to a single control section whose length is the maximum length specified in the CM items of that name (or blank). No text can be contained in a common control section.
- Within an input module, the binder does not accept an SD or PC entry after the first RLD item is read.

To avoid unnecessary scanning and input/output operations, input modules should conform with the following conventions. Although violations of these rules are not treated as errors, avoiding them will improve the efficiency of binder processing.

- Within an object module, no LD or SD can have the same name as an ER.
- Within an object module, no two ERs can have the same name.
- Within an object module, TXT records can be in the order of the addresses assigned by the language translator.

Record formats

The following figures show the record formats required for object modules processed by the binder.

SYM record

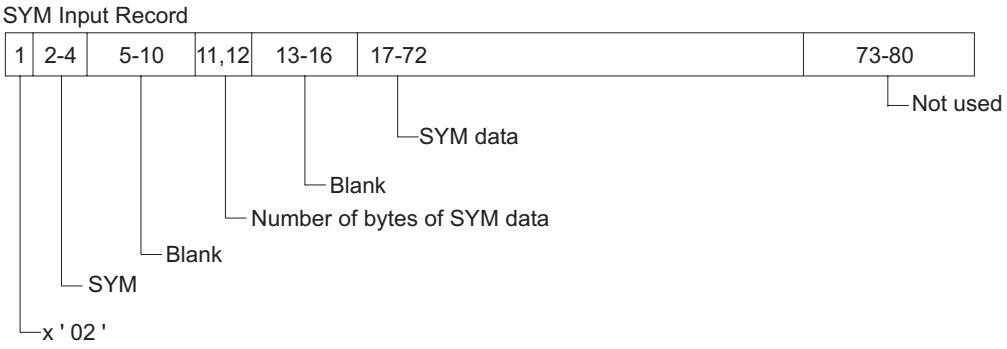


Figure 6. SYM input record

ESD record

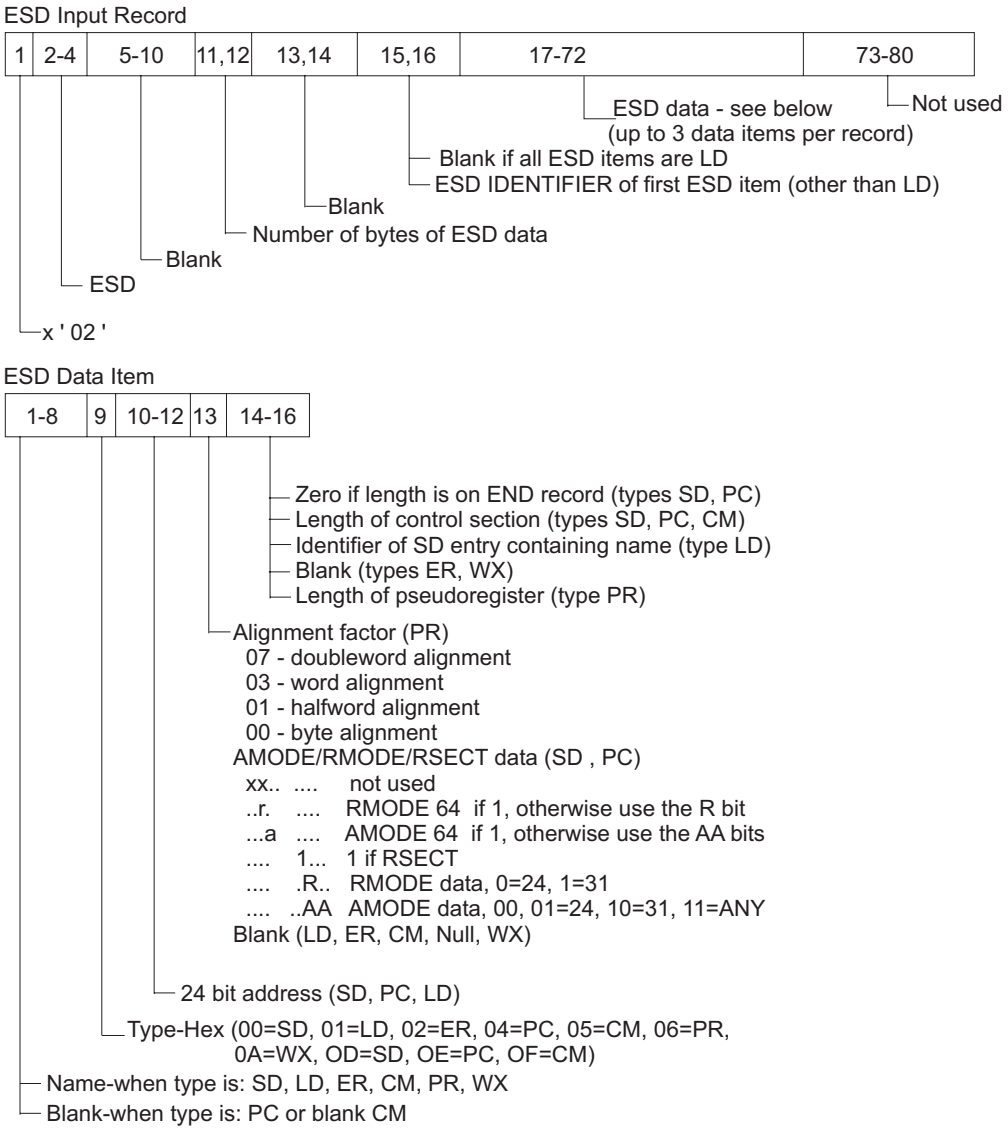


Figure 7. ESD input record

Note: In 0D=SD, 0D is the quadword version, in 0E=PC, 0E is the quadword version, and in 0F=CM, 0F is the quadword version.

Text record

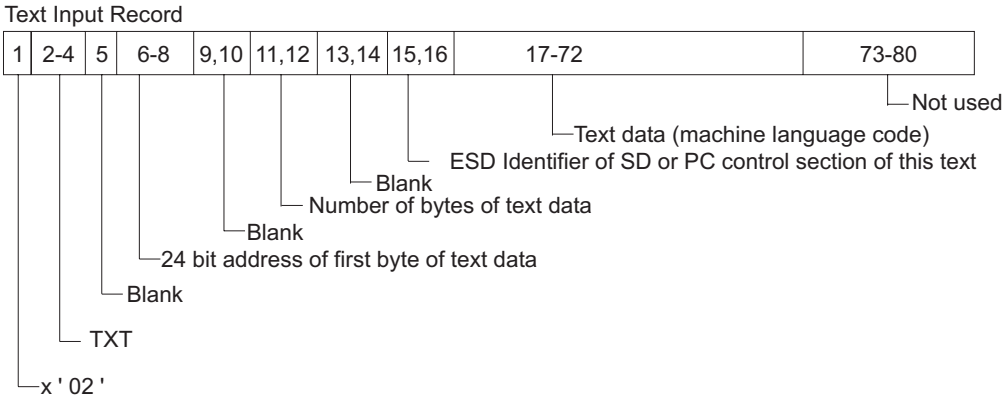


Figure 8. Text input record

RLD record

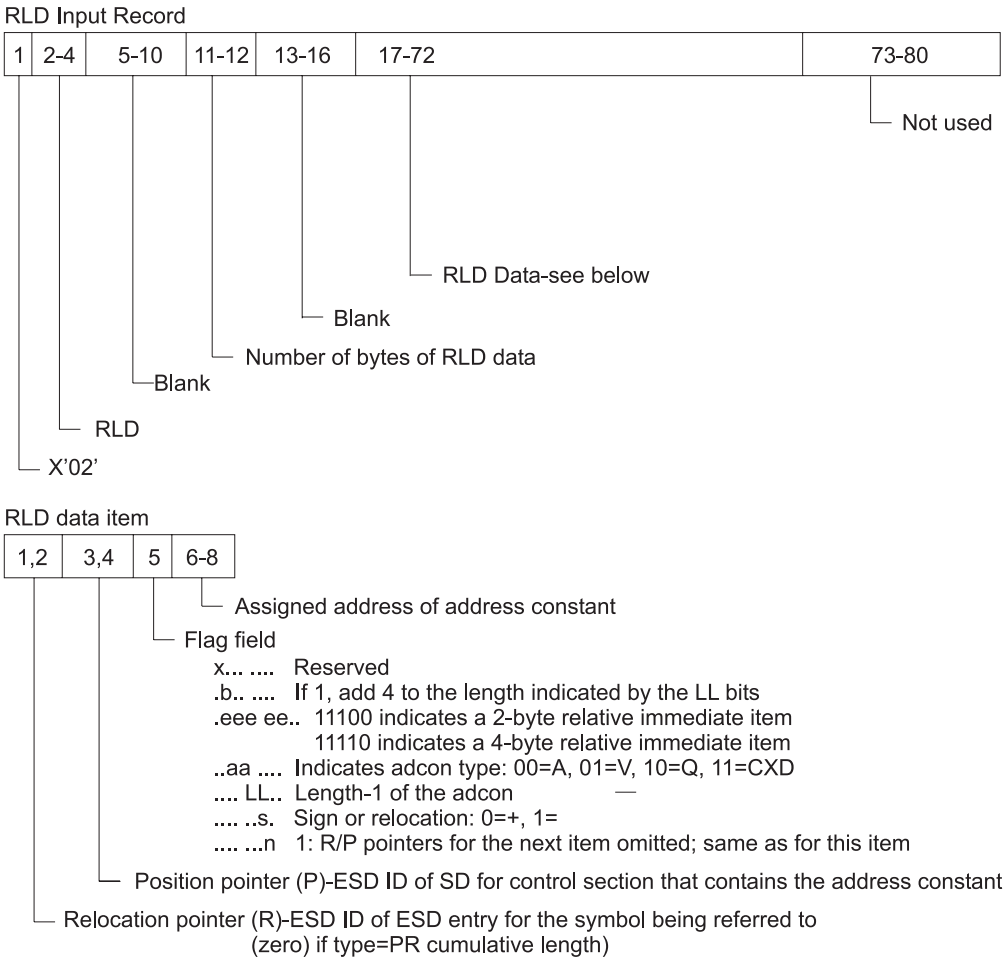


Figure 9. RLD input record

END record

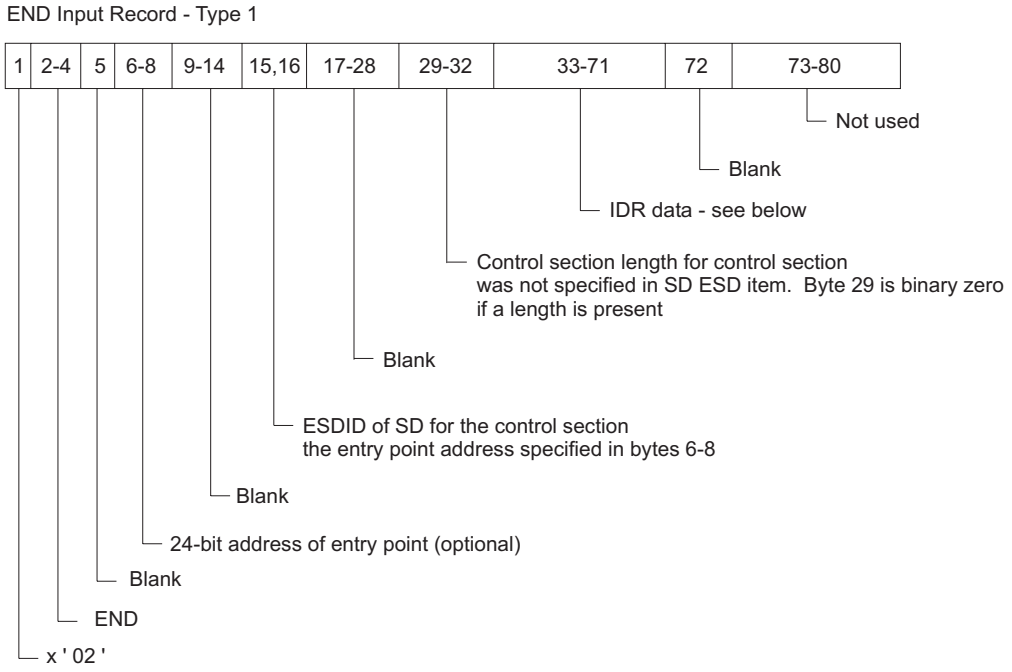


Figure 10. END input record–type 1

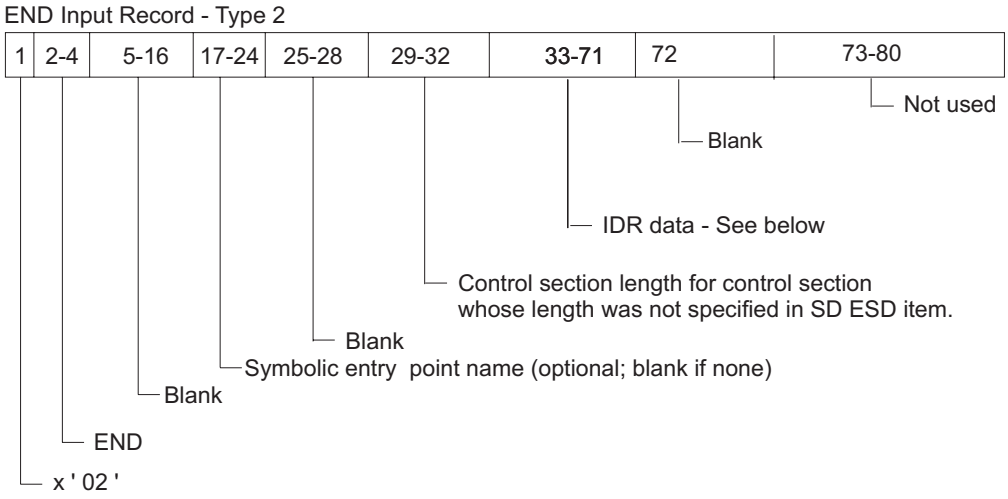
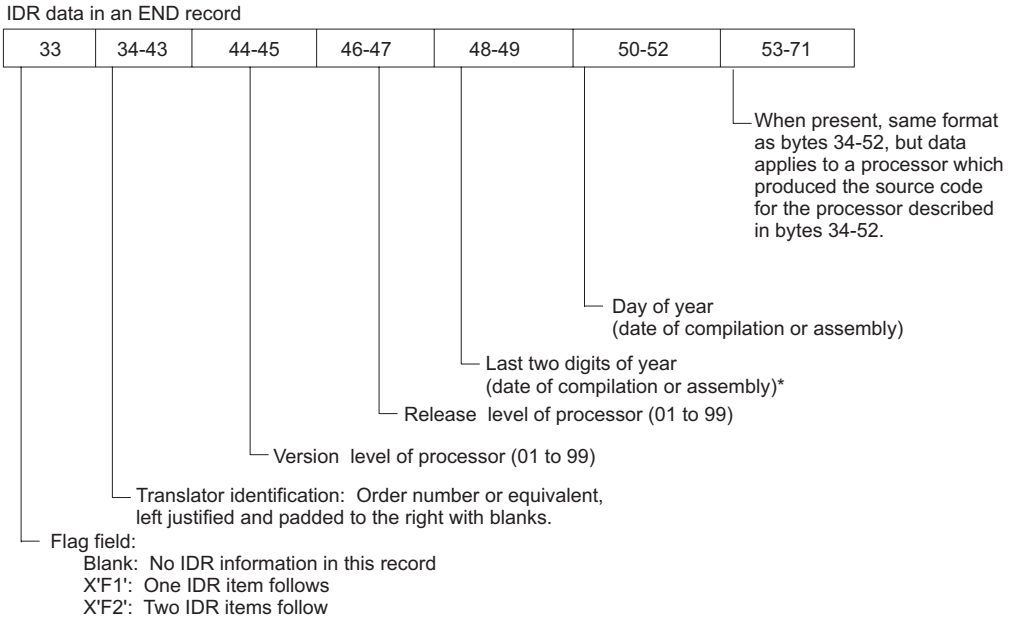


Figure 11. END input record–type 2



Note: * 65-99 represents 19XX years and 00-64 represents 20XX years.

Figure 12. IDR data in an object module END record

Extended object module (XOBJ)

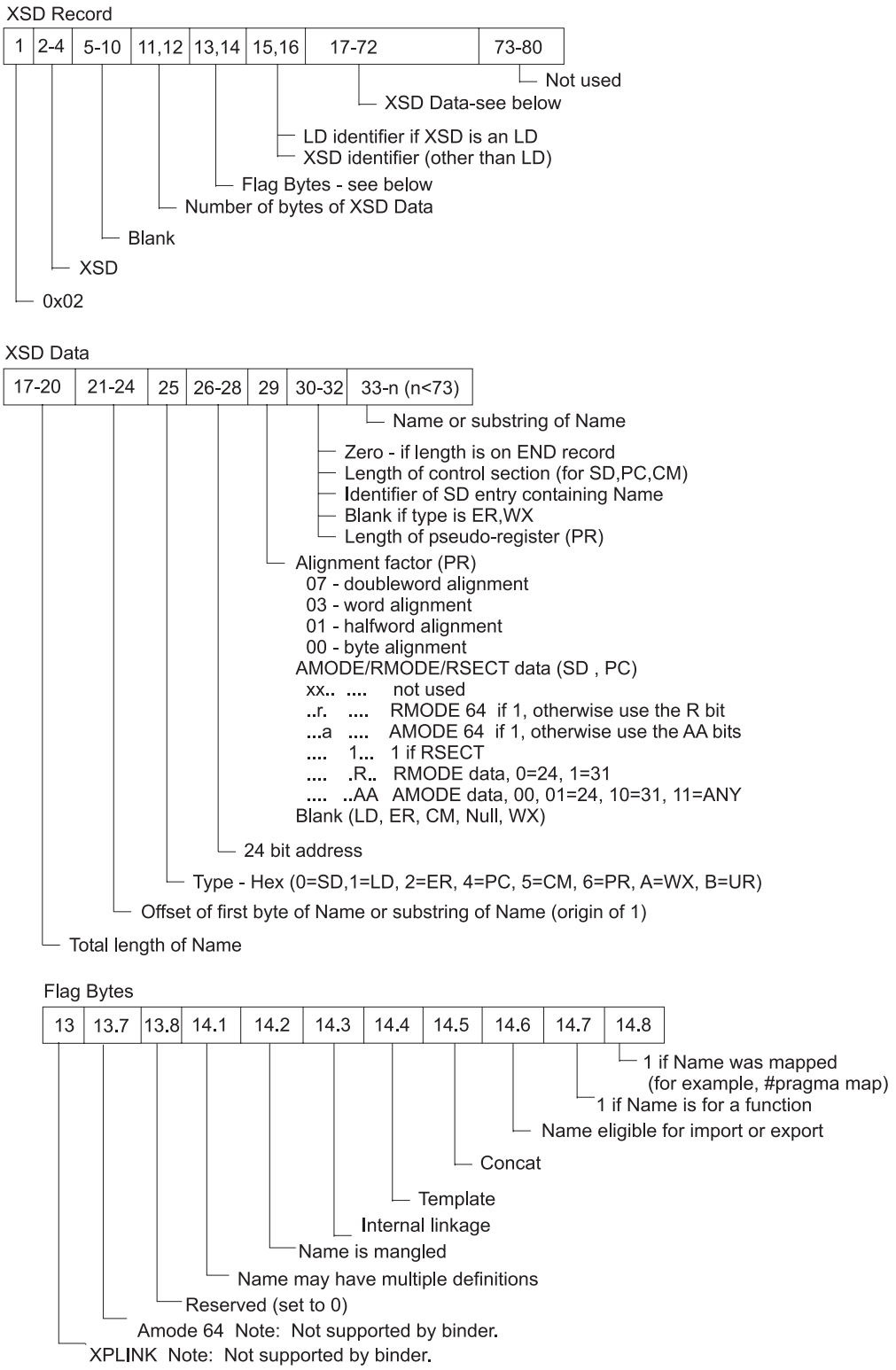
Some compilers may create a modified object module. The z/OS C compiler, for example, may create this modified object module format when LONGNAME or RENT is in effect as a compile option. The modified object module consists of several subfiles, each of which has the form of an ordinary object module, except that the ESD records will generally be replaced by XSD records. The first subfile consists of two records. The first record is an ESD record containing a single ESD item: a 0-length section definition (SD) for the name @@DOPLNK. The second record is an END record with the appropriate product and date information. It is this subfile which triggers the recognition of this module as an XOBJ module.

XOBJ modules can be processed by either the Language Environment prelinker or the binder. XOBJ modules (input to the binder) which contain an XSD record for symbol @@XINIT@ can be saved only in a PO3 (or greater) format program object. XOBJ modules containing an XSD record for @@XINIT@ are those which use C-style reentrancy.

When XOBJ input modules are saved in a PO3 or higher format program object, the sections in the program object will not always correspond to CSECTS defined by SD records in the input module. CSECTS with compiler-generated names (those beginning with @@) may be renamed, or combined with other CSECTs into a single output section. The @@DOPLNK csect will not appear in the output module.

The remainder of this appendix defines the XSD record format.

XSD record format



Notes

1. The record is structured like the ESD record. This is so that it is simple to map a XSD record to an ESD record. Differences from the ESD record, and other details, are described here.
2. The length of a name can be stored in a four byte field. A name greater than 40 characters in length is spread over several records.
3. For a name spread over several records, all of the information is repeated on each record except for the 'Offset' and 'Substring of Name' field.
4. The LD identifiers [2] and XSD identifiers can collide. That is, LD identifiers and XSD identifiers are different identifier namespaces.
5. PC sections have a name length of 0.
6. the 'if Name was mapped' bit is set for:
   - names that are mapped using #pragma map
   - control section names specified using #pragma csect
   - reserved run-time names generated by the compiler.

2. A LD does not have an identifier field on an ESD record. This field is introduced to tie together all the XSD records for a given name.

Appendix B. Load module formats

This topic contains general-use programming interface and associated guidance information.

This topic contains load record formats (see Figure 13 through Figure 21 on page 202).

See also “Extended object module (XOBJ)” on page 191 for the format used for LONGNAME support for the C compiler. Appendix C, “Generalized object file format (GOFF),” on page 203 discusses support for the generalized object file format.

Input conventions

Load modules to be processed in a single execution of the binder must conform with a number of input conventions. Violations of the following are treated as errors by the binder:

- All the ESD records must precede the text records.
- The end of every load module must be marked by an EOM flag.
- RLD items must appear after the text record containing the adcons which they describe.
- All SYM records must be placed at the beginning of the load module.
- During a single execution of the binder, if two or more control sections having the same name are read in, only the first control section is accepted; the subsequent control sections are deleted
- The binder interprets common (CM) entries in the ESD (blank or with the same name) as references to a single control section whose length is the maximum length specified in the CM items of that name (or blank). No text can be contained in a common control section

Record formats

Figure 13 through Figure 19 on page 200 are the load module record formats for the linkage editor.

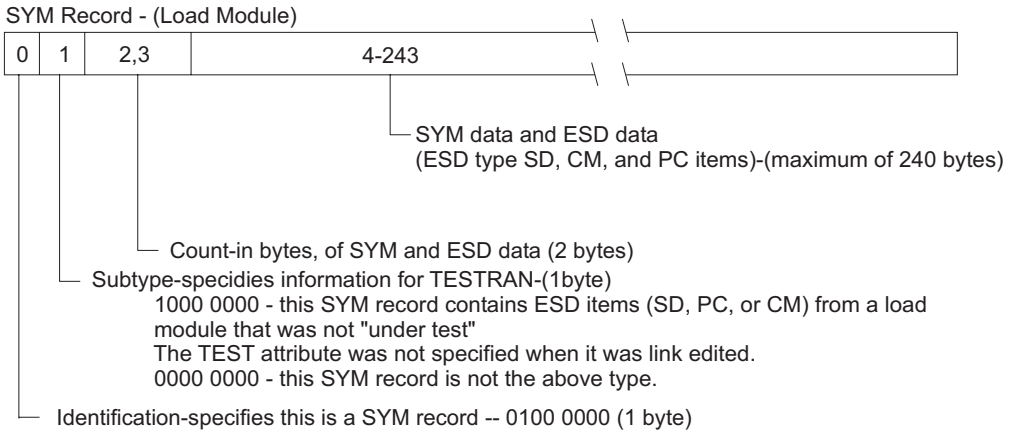


Figure 13. SYM record (load module)

© Copyright IBM Corp. 1991, 2012

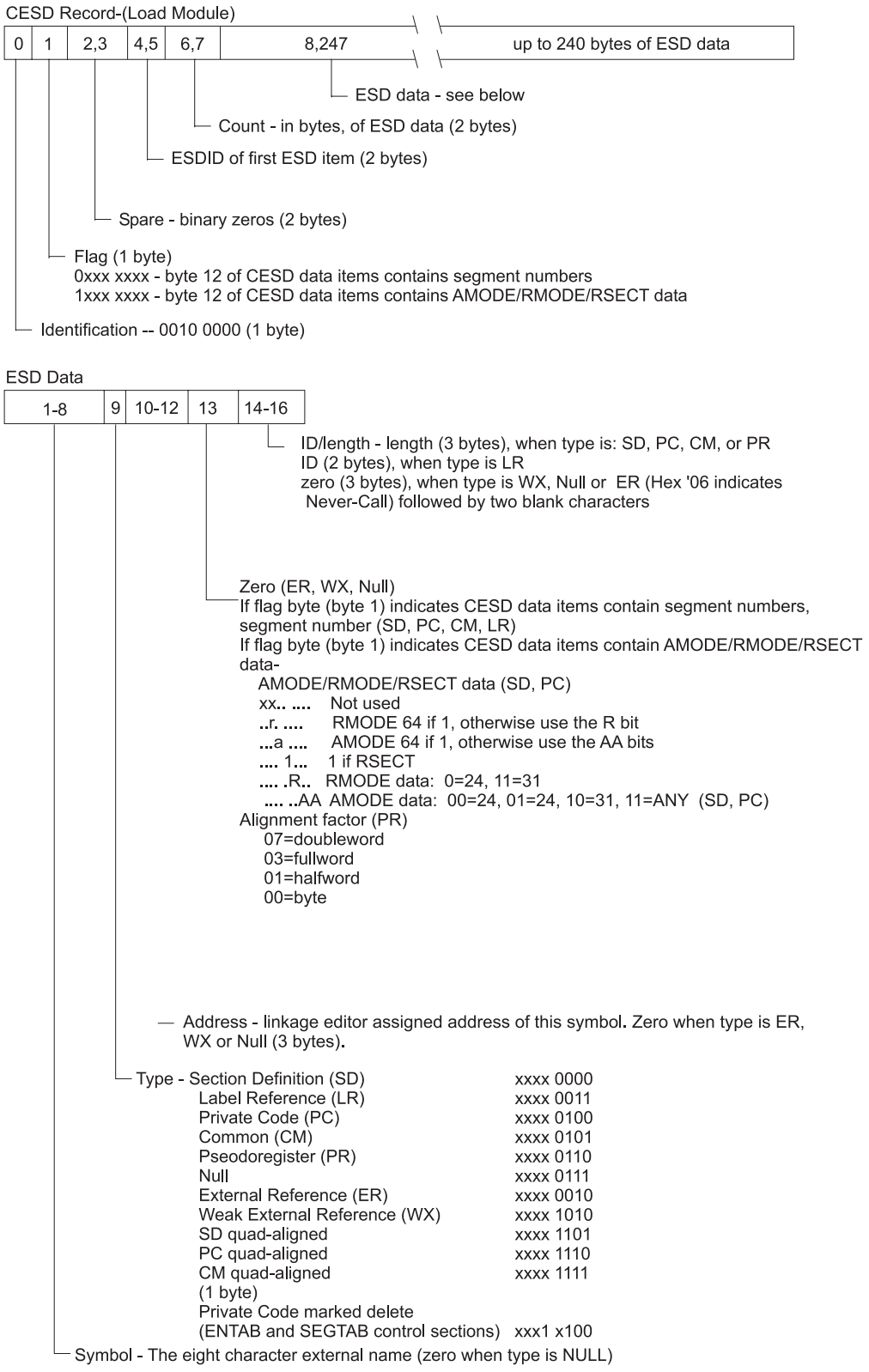


Figure 14. CESD record (load module)

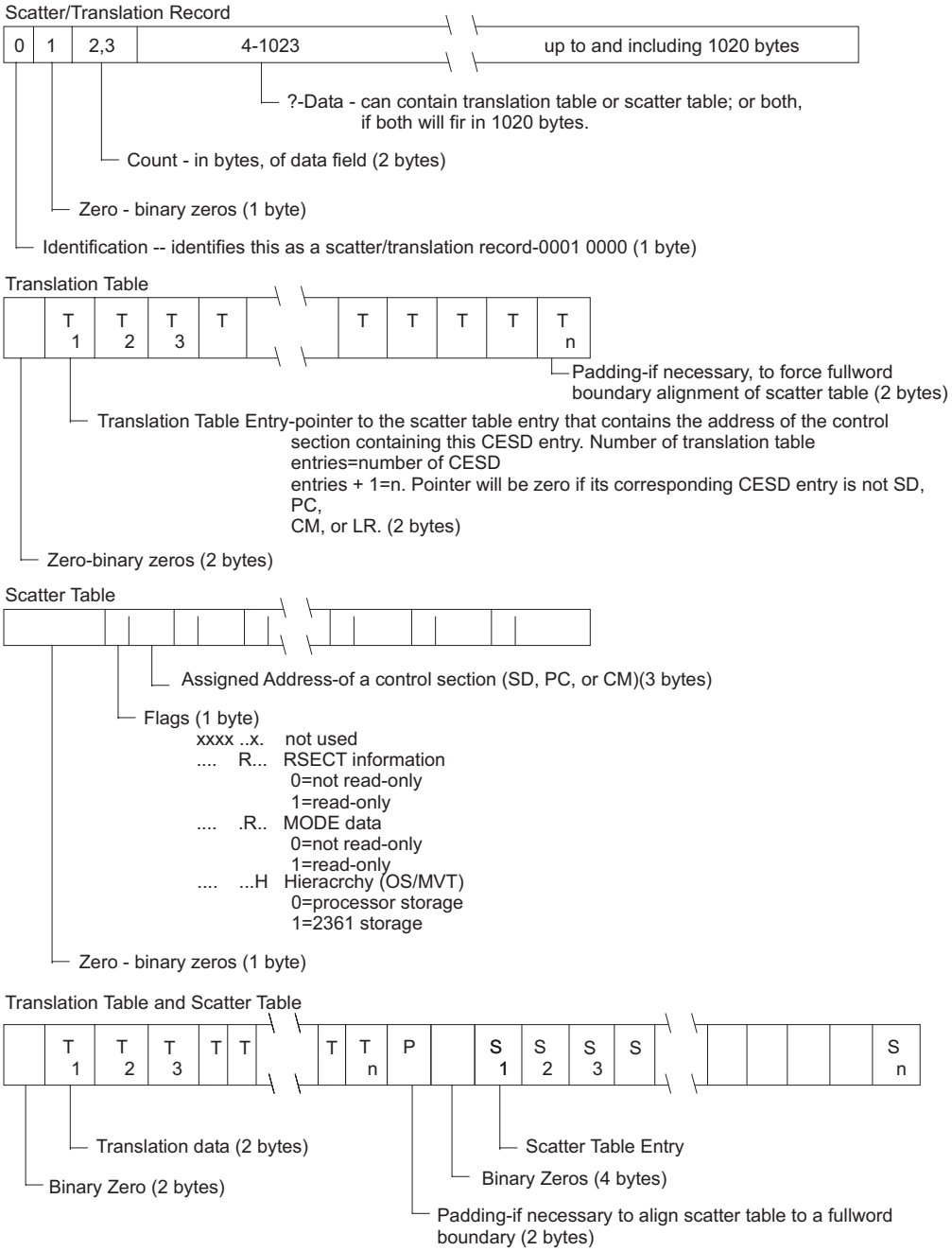


Figure 15. Scatter/Translation record

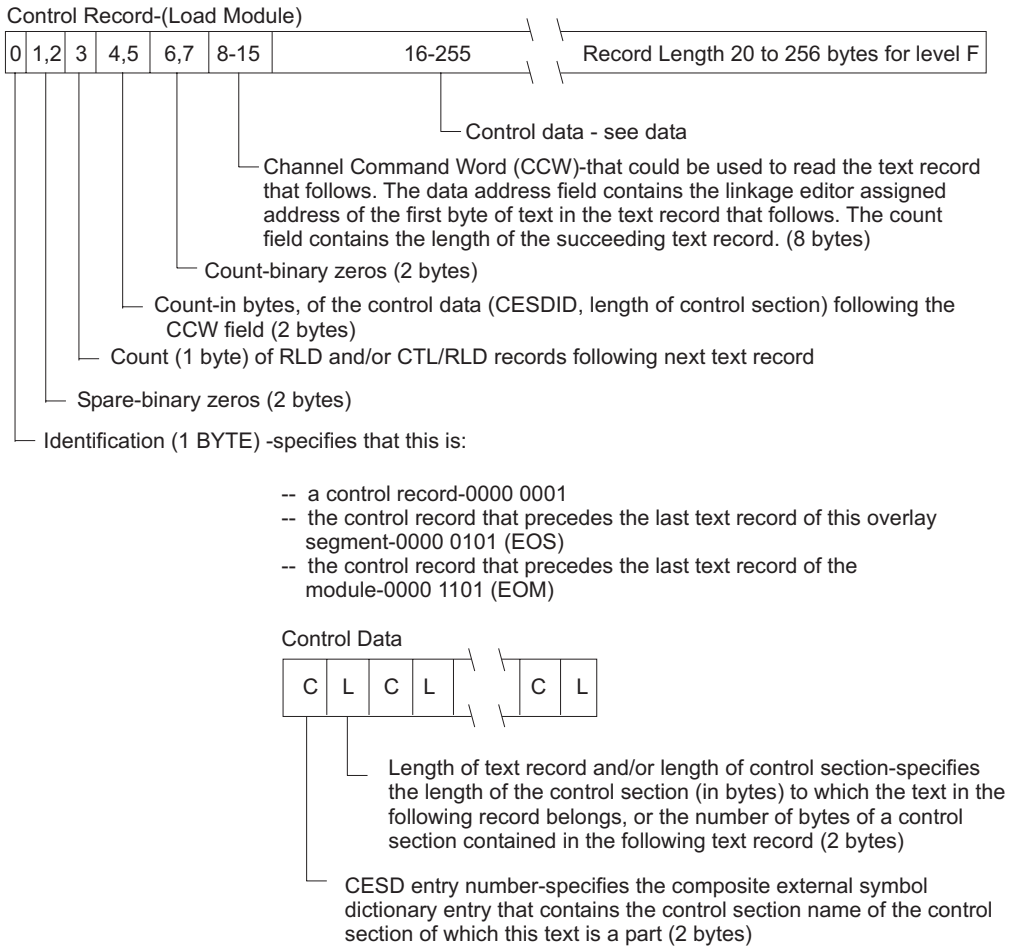


Figure 16. Control record (load module)

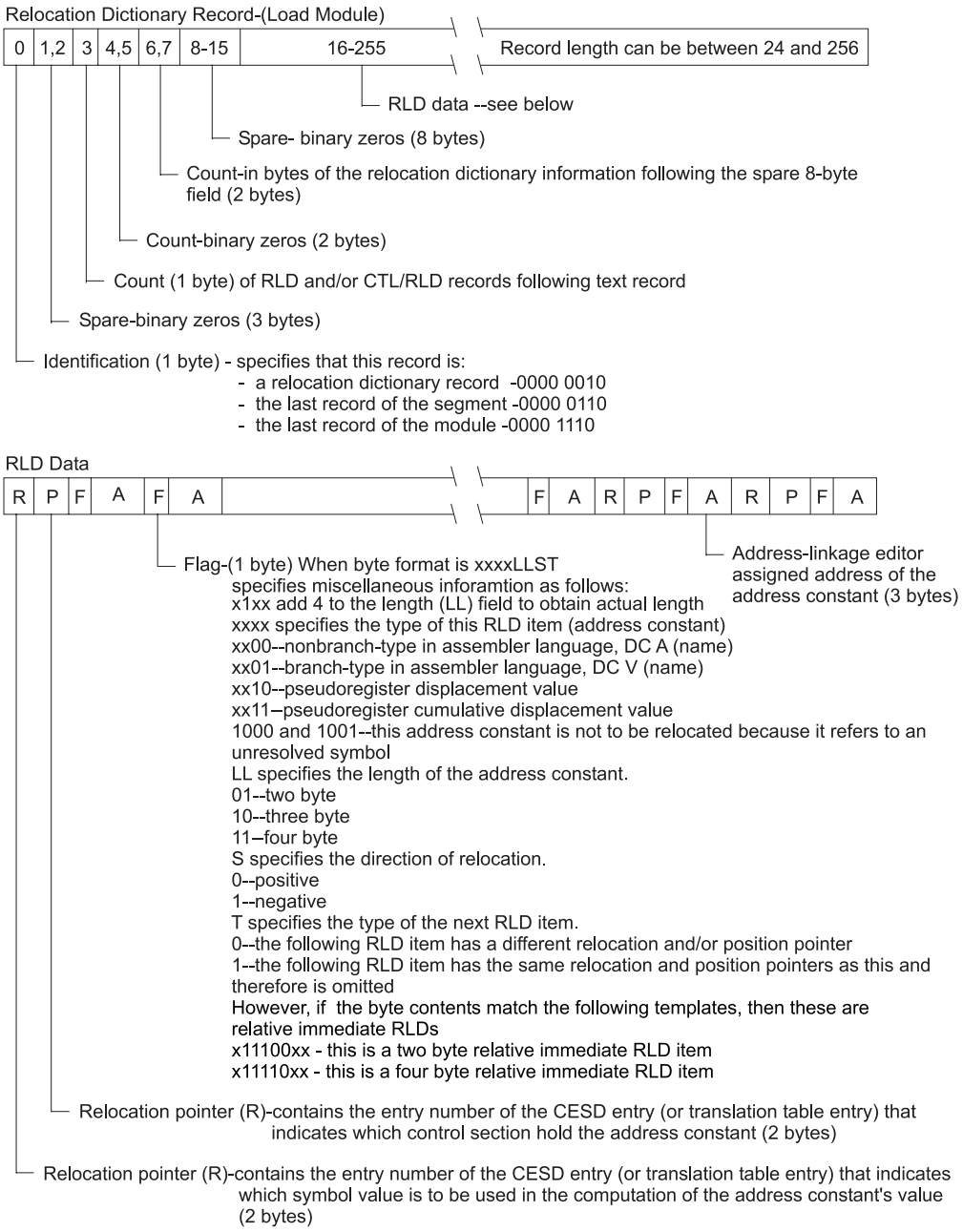


Figure 17. Relocation dictionary record (load module)

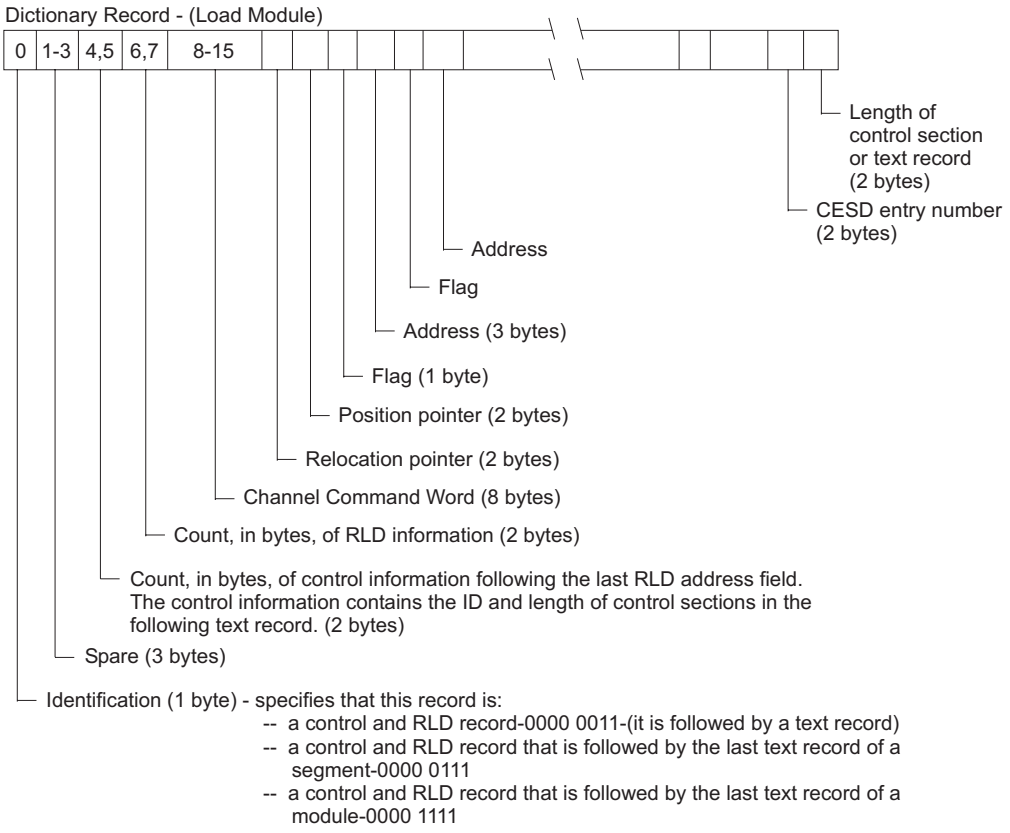


Figure 18. Control and relocation dictionary record (load module)

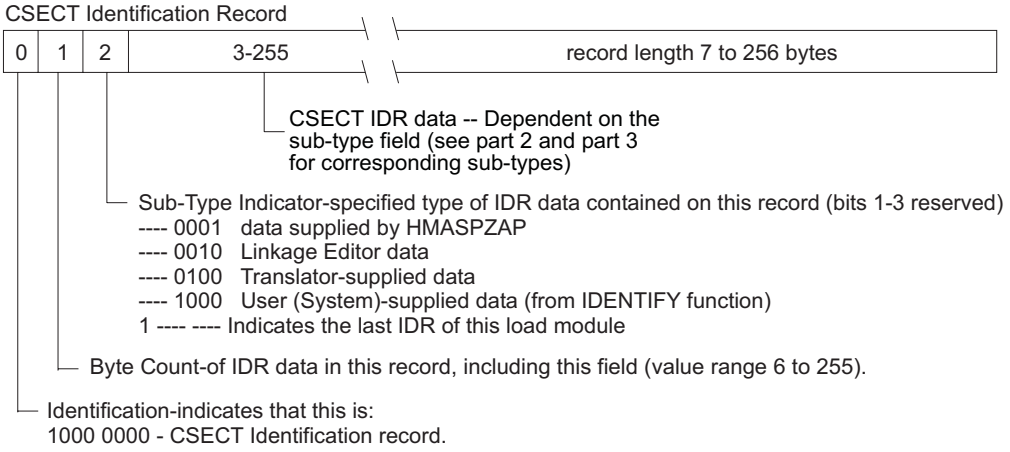
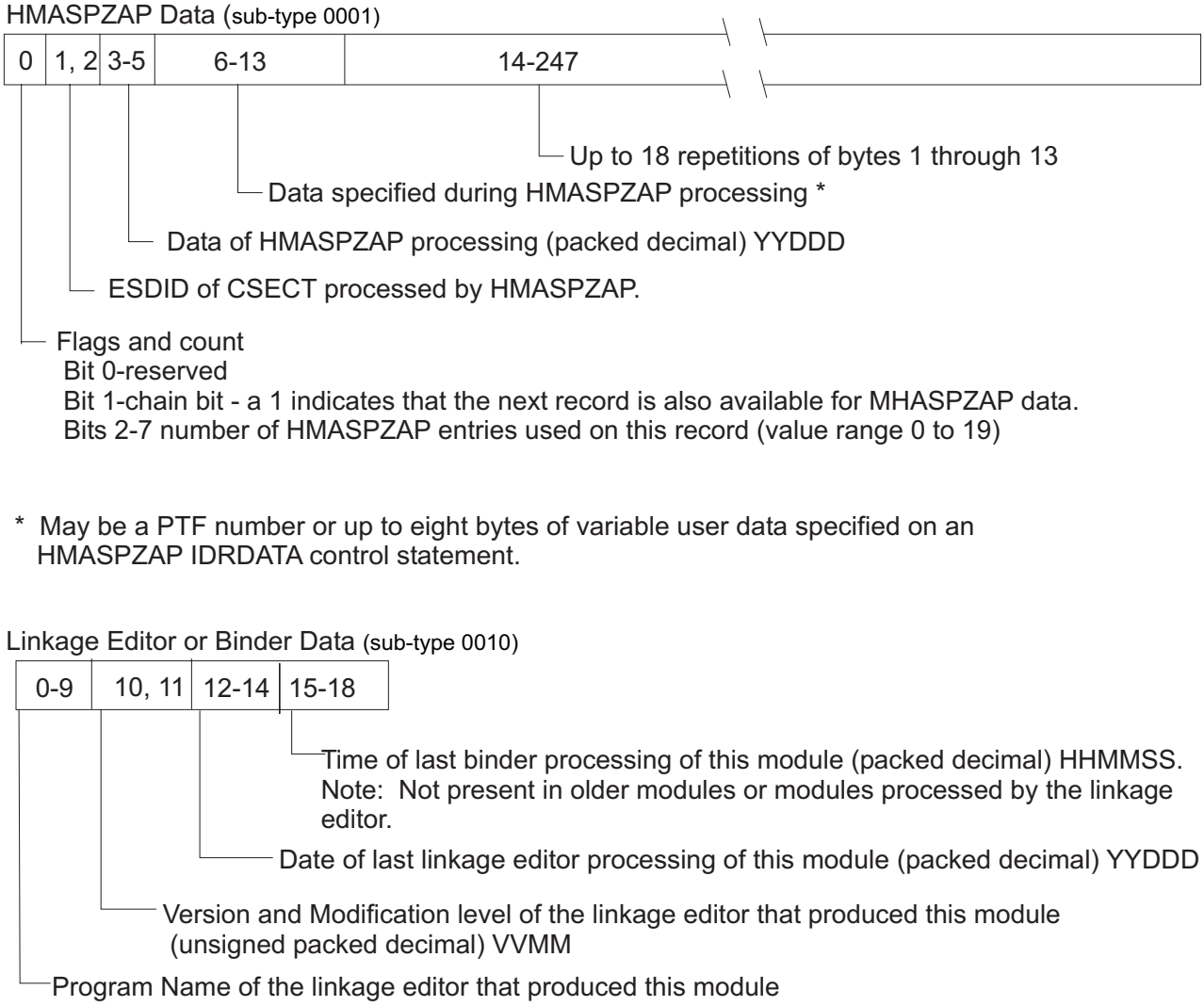


Figure 19. Record format of load module IDRs–part 1



* May be a PTF number or up to eight bytes of variable user data specified on an HMASPZAP IDRDATA control statement.

Note: Date and time fields contents of "65001F0000000F" are generated by the binder when copying a module lacking a binder IDR record.

Figure 20. Record format of load module IDRs–part 2

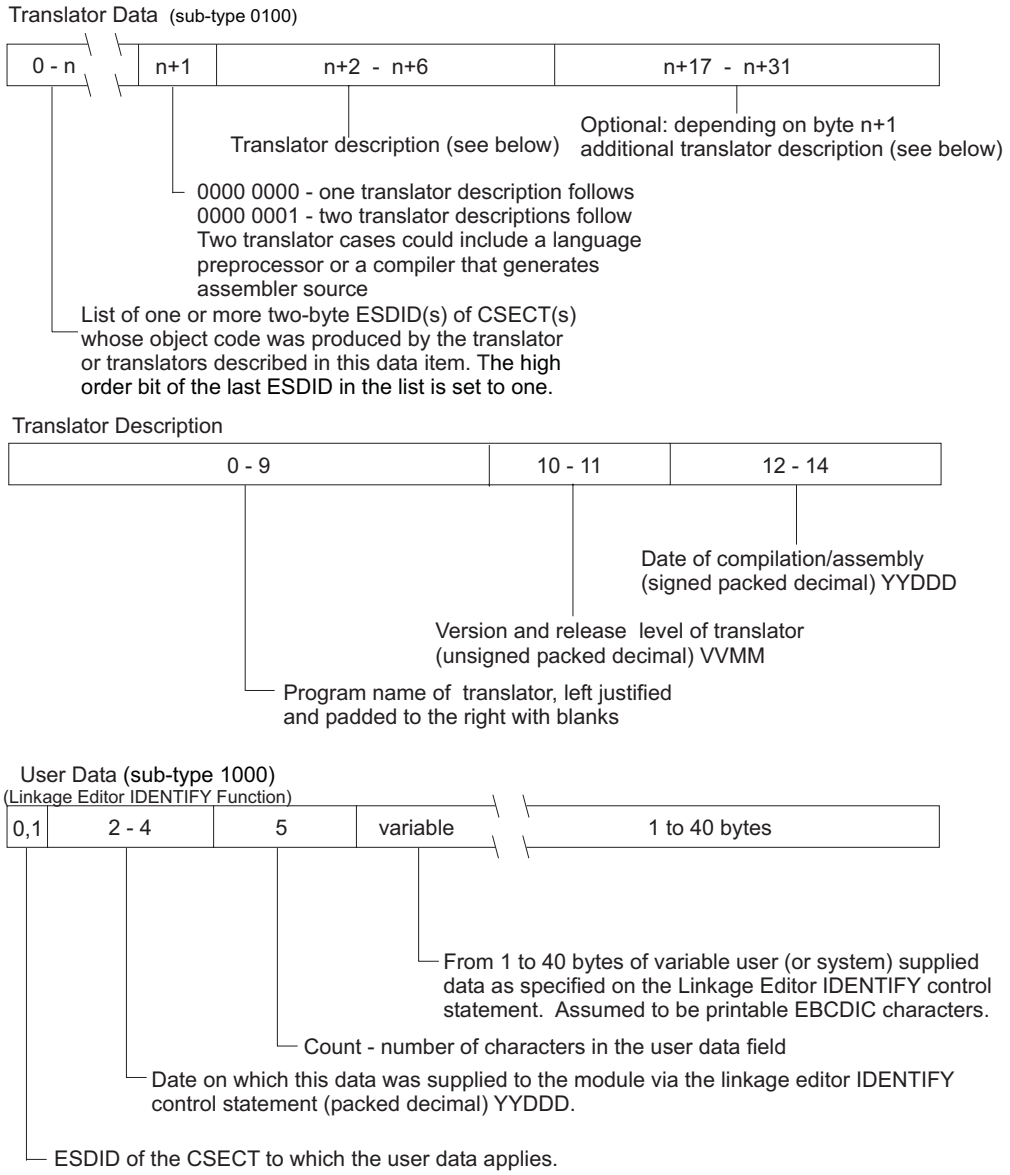


Figure 21. Record format of load module IDRs–part 3

Appendix C. Generalized object file format (GOFF)

This topic contains general-use programming interface and associated guidance information.

This topic describes the conventions and formats for the generalized object file format (GOFF). The data elements supported by the binder are shown here.

The GOFF record types are essentially identical in concept (and usually, in content) to OBJ (object module) records, but their formats differ in several respects. Migration from OBJ to GOFF formats is discussed in “Mapping object formats to GOFF format” on page 230. In the remainder of this topic, the term OBJ refers to traditional object formats, including the extended object format.

The term “record” is used in two possibly conflicting senses throughout this topic. The first sense is that of one of the GOFF record types, regardless of the number of data management records it may be spread over. The second sense is that of a data management “logical” record, which contains one of the pieces of a continued GOFF record. The correct sense should be clear from the surrounding context.

The term “object file” is used to refer to a collection of GOFF records beginning with a header record and ending with an end record.

Guidelines and restrictions

GOFF records follow these guidelines and restrictions:

- A GOFF record is contained within one or more complete data management records. These must be fixed or variable length records, with only fixed length records supported in a UNIX file system. If fixed length records are used, the record length must be 80. Because continuation records are not defined for header records, the minimum usable LRECL for variable length records is 58.
- A record's length is *not* a part of the record itself. This architecture assumes that the operating environment in which the records are produced will supply additional information, such as record length (LRECL) and record format (F or V). This means that a program reading GOFF files need not know the internal structure of each record in order to scan for a particular record type.
- Each GOFF record must start at a data management record boundary, that is, a GOFF file is not a stream. When fixed length records are used, including the UNIX file case, some records will need to have unused space at the end to meet this requirement.
- GOFF records contain no sequence information. The sequence of the records in a GOFF file cannot be altered. Continuation records must appear in the correct sequence, immediately following the record they continue.
- Each GOFF object file must begin with a header record, and end with an end record. (Multiple object files can be produced in a single invocation of a translator, but the object stream must be separable into distinct object files.) The “Record Count” field on the END record should contain a count of the GOFF records (not data management logical records) in each object file including HDR and END.
- If a GOFF record is continued, no intervening records can appear among the continuation segments.
- Some of the OBJ conventions for ordering and contents of the input records apply. Specifically:

© Copyright IBM Corp. 1991, 2012

  - ESD items must be numbered starting at 1 in a monotonically increasing sequence, with no “gaps” or omitted values.
  - An ESD record must precede any other record containing a reference to the ESDID defined by that item.
- Multiple RLD data items can appear on a single RLD record, and RLD items can be split across continuation records.
- ESD records contain only a single ESD item.
- The character set to be used for external names is not specified in this GOFF format. Other products, standards, or conventions can impose restrictions, of course. The program management binder requires that the characters in external symbols lie in the range from X'41' to X'FE' inclusive, plus the Shift-Out (X'0E') and Shift-In (X'0F') characters.

  The character set used in each GOFF file, including the character set used for external names, can be specified in the Module Header record.
- For upward compatibility, fields that are shown as reserved should be set to binary zero. They might be supported by a future version of the binder.
- If architectural extensions cause a record definition to be longer than was defined at the same or an earlier architecture level, the binder treats the additional fields as containing binary zero if they are not actually present in the record.

Note: Basically, the program management binder supports any EBCDIC Single-Byte Character Set (SBCS), plus SBCS-encoded Double-Byte Character Sets (DBCS). The main reason for including a character set identifier in the GOFF header record is to prepare for support of Multiple-Byte Character Sets (MBCS) such as Unicode.

Incompatibilities

The following usages of GOFF records are incompatible with OBJ records.

- OBJ SYM records have no direct counterpart in GOFF records. Other more flexible vehicles (particularly, ADATA records) must be used instead.
- Some users have found unsupported uses of OBJ records such as hiding data in otherwise “blank” or “unused” fields. Such uses are not supported for GOFF records.

GOFF record formats

The basic GOFF record types are shown below. The names used here are similar to OBJ record names, and have similar functions. The one new type, LEN, has a name with similar mnemonic significance. These “names” do not appear anywhere on the records, however (unlike OBJ records).

HDR Header Record: must appear first.

ESD External Symbol Definition

TXT Machine language instructions and data (“text”); Identification Data Record (“IDR”) data; Associated Data (“ADATA”); other identification data (more than current 19-byte IDR items); any other data items to become part of the module. (Note that “text” is used here as a very general term; see the definition in the glossary).

RLD Relocation Directory (adcon information)

LEN Supply length values for deferred-length sections

Note: A LEN record is essentially a special form of ESD record, in which only a single piece of data, a length, is provided for a previously defined external name.

END End of module, with optional entry-point identification. The END record must be the last record in the object file.

Conventions

Several sets of conventions are to be followed in using and describing GOFF records.

In general, each record has at most one varying-length field.

Conventions for record descriptions

The fields in the records are described using four columns:

- A short phrase or name describing the field
- The offset of the field from the start of the record, or from the start of the record component. Numbering is zero-based, and bits within a byte follow the standard System/390® convention where the high-order (leftmost) bit in a byte (with the greatest numeric weight) is bit zero. The notation is one of these forms:
  - **n** Byte number n
  - **n-m** Bytes numbered n through m
  - **n-*** Bytes starting at n, through the end of the field or record
  - **n.p** Byte number n, bit number p
  - **n.p-q** Byte number n, bits numbered p through q
- The length of the field is described as follows:

  Byte(n)
  A contiguous string of n bytes, with no particular numeric value assigned, nor any significance for particular bits within the bytes.

  Binary(n)
  A contiguous string of n bytes, to be treated as a signed two's-complement binary integer. (If it is interpreted as an unsigned integer, the field description will explicitly state this.)

  Bit(n) A contiguous string of n bits.
- The contents of the field.

Conventions for class names

Many of the GOFF records require specifying a class name. Some of these names can be chosen freely (for example, for user-supplied information), and other names must follow some rules. Class names reserved to IBM are called “Reserved Names”, and class names that can be used freely are called **Non-Reserved Names**. (Note that class names are *not* external symbols!)

- Class names appearing in object modules are case sensitive.
- **Reserved** class names are a maximum of 16 characters and are of the following form:

  - The first character of a reserved class name is a letter (A-Z).
  - The second character of a reserved class name is an underscore character (_).

  - The remaining 1 to 14 characters of a reserved class name form a normal alphanumeric identifier: a letter followed by either letters, digits or both.

  The following reserved class name prefixes are currently in use:

  - Class names starting with B_ are reserved for the binder.
  - Loadable class names using the prefix C_ are reserved for translators creating Language Environment-enabled code.
  - X_,Y_,Z_ as well as non-loadable C_ classes are reserved for general translator use.

  All other *letter_* class names are reserved for future use.

- The following classes can be specified by language translators and other creators of GOFF records:

  ```
  B_ESD          External Symbol Dictionary Class
  B_TEXT         Text Class
  B_RLD          Relocation Directory Class
  B_SYM          Internal Symbol Table Class
  B_IDRL         Language-Translator Identification Data Class
  B_PRV          Pseudo-Register Class
  B_IDRU         User-specified Identification Data Class
  ```

  OBJ inputs to the program management binder will be automatically mapped to these classes.

  The following classes are reserved for internal use by the program management binder, and cannot be specified on GOFF records:

  ```
  B_IDRB         Program Management Binder-created
                 Identification Data Class
  B_IDRZ         SuperZAP-created Identification Data Class
  B_MAP          Internal Program Object Mapping-Data Class
  B_LIT          Loader Information Table
  B_IMPEXP       Import-Export Table
  ```

- It is expected that conventions will be established for the use of other classes. IBM translators should use class names of the form:

  ```
  C_xxxxxxxxx    Translator-defined Classes
  ```

- **Non-Reserved** class names follow the normal formation rules for external symbols except they are limited to 16 characters in length.

Record prefix

All GOFF records have a 3-byte identifying prefix of the following form:

Table 48. GOFF record-type identification prefix byte

Field	Offset	Type	Description
Prefix	0	Byte(1)	X'03'. All other values are reserved. **Note:** The value X'03' is chosen to distinguish GOFF records from OBJ records, which begin with a byte containing X'02', and from control statements, which begin with a byte whose hex value is not less than X'40'.

Table 48. GOFF record-type identification prefix byte (continued)

Field	Offset	Type	Description
Record Type	1.0-3	Bit(4)	Type of record: **X'0'** External Symbol Dictionary (ESD) **X'1'** Text (TXT) **X'2'** Relocation Directory (RLD) **X'3'** Deferred Element Length (LEN) **X'4'** Module End, with optional entry point specification (END) **X'5-E'** Reserved. **X'F'** Module Header (HDR)
	1.4-5	Bit(2)	Reserved.
Continuation and Continued Indicators	1.6-7	Bit(2)	These two bits indicate respectively that this record is • a continuation of the preceding record, and/or • continued on the succeeding record. **B'00'** This record is the initial record of the designated type (it is not a continuation record). It is not continued on the succeeding record. **Note:** For variable-format GOFF records, this should be the only valid combination. **B'01'** This record is the initial record of the designated type (it is not a continuation record). It is continued on the succeeding record. **B'10'** This record is a continuation of the previous record of this type, and it is not continued on the following record. **B'11'** This record is a continuation of the previous record of this type, and it is continued on the following record.
Version	2	Byte(1)	**X'00'** This is the initial version number for the record of this type. All other values are reserved.

These three bytes are usually referred to as the “PTV” bytes.

Module header record

A module header (“HDR”) record describes global properties of this GOFF file. It is required, and must appear first.

Table 49. Module header record

Field	Offset	Type	Description
PTV	0-2	Byte(3)	Type of record **X'03F000'** If not continued (this is the only form currently defined)
	3-47	Byte(45)	Reserved.
Architecture level	48-51	Binary(4)	GOFF architecture level used in this object file. Only levels 0 and 1 are supported.
Module properties	52–53	Binary(2)	Length of module properties field length.

Table 49. Module header record (continued)

Field	Offset	Type	Description
	54–59	Byte(n)	Reserved.
Module properties	60–	*	Module properties field.

External symbol definition record

An external symbol definition ("ESD") record describes a single symbol. An initial record has the following form:

Table 50. External symbol definition record

Field	Offset	Type	Description
PTV	0-2	Byte(3)	Type of record **X'030000'** If not continued **X'030100'** If continued
Symbol Type	3	Binary(1)	Type of the external symbol: **X'00'** SD, section definition. This will include current OBJ ESD items of types SD, CM and PC (private code, blank SD name). The "Parent" or "Owning" ID must be zero. **Note:** PC sections are defined by having a blank section name, not by a special section type (see "Mapping object module ESD PC items" on page 232). See the comments in the description of the COMMON flag in Table 55 on page 216. **X'01'** ED, element definition. The "Parent ID" must be nonzero, and is the SD ID of the section of which this element is a component. **Note:** The class to which this element belongs is specified by the external name on this record. If any RLD items refer to this class, an ED record must be present, even if the class contains no text. **X'02'** LD, label definition. The "Parent ID" must be nonzero, and is the EDID of the element in which the LD item resides. **X'03'** PR, part reference. The "Parent ID" must be nonzero, and is the EDID of the element in which this PR item resides. **Note:** This supersedes current OBJ PR (pseudo-register definition). **X'04'** ER and WX, external reference. (WX is described by the "Binding Strength" value in the Attributes field below.) All other types are reserved.

Table 50. External symbol definition record (continued)

Field	Offset	Type	Description
ESDID	4-7	Binary(4)	ESDID of this item. This value must be greater than zero. ESDIDs in a GOFF file must be numbered sequentially starting at one, with no gaps.
Parent or Owning ESDID	8-11	Binary(4)	ESDID of the object that "defines" or "owns" this item, if any. (For example, an LD item has its own ESDID, but the element to which it belongs has the "owning" ESDID.) If there is no "owning" ESDID, this field contains binary zero. See Table 52 on page 211 for a summary.
	12-15	Binary(4)	Reserved.
Offset	16-19	Binary(4)	Offset associated with the ESD item named on this record. **Note:** The only ESD types currently having offsets are LD items (and ED items as currently produced by the Assembler).
	20-23	Binary(4)	Reserved.
Length	24-27	Binary(4)	Length of the item named on this record. **Note:** Length values apply only to ED and PR items. For SD, LD, and ER items, this field must be zero. To distinguish a true zero length from a deferred length, specify –1 if the length specification is deferred. See "Deferred element length record" on page 227.
Extended Attribute ESDID	28-31	Binary(4)	ESDID of the element that contains extended attribute information. This information is stored in the program object and may be made available to Language Environment DLL support for dynamically resolved symbols. Valid for ED and LD records.
Extended Attribute Data Offset	32-35	Binary(4)	Offset (within the element defined by the previous field) of the extended attribute data defined for this symbol. Valid for ED and LD records.
	36-39	Byte(4)	Reserved.
Name Space ID	40	Binary(1)	Identification of the name space to which this name belongs. **0** Reserved to program management binder **1** Normal external names (these can be specified as PDS member and alias names) **2** Pseudo-Register (PR) names **3** Parts. External data items for which storage is allocated in the program object and which can have initial text (for example, data items in C writable static). All other values are reserved.
Fill-Byte Value Presence Field	41.0	Bit(1)	Indicates the presence of a fill-byte value in Byte 42. **B'0'** No fill byte is present. **B'1'** A fill byte value is present. Valid only for ED records.

Table 50. External symbol definition record (continued)

Field	Offset	Type	Description
Mangled Flag	41.1	Bit(1)	Indicates whether this symbol is a C++ mangled name. **B'0'** The symbol is not mangled. **B'1'** The symbol may be mangled.
Renameable Flag	41.2	Bit(1)	Indicates whether or not the symbol can participate in Language Environment type renaming. **B'0'** The symbol cannot be renamed (same as 'mapped' flag in XOBJ). **B'1'** The symbol can be renamed.
Removable Class Flag	41.3	Bit(1)	Indicates that this class may be optionally deleted from a program object without affecting the executability of the program. **B'0'** Not removable. This is the default. **B'1'** Removable. This attribute applies only to ED ESD items.
	41.4-6	Bit(3)	Reserved.
	41.7	Bit(1)	**B'1'** Reserve 16 bytes at beginning of class. MRG class ED records only. **B'0'** No space reserved.
Fill Byte Value	42	Byte(1)	Fill byte value, if bit 41.0 is one. ED records only.
	43	Byte(1)	Reserved.
Associated data ID	44-47	Binary(4)	This ID identifies the associated data (the environment or static area) for this symbol. Valid for LD items only. Supported only for XPLINK definitions and references.
Priority	48-51	Binary(4)	Supported for PR items only. This unsigned field is used as a sort field when determining the order of PR items.
	52-59	Byte(8)	Reserved. Must be binary zero.
Behavioral Attributes	60-69	Byte(10)	Behavioral attribute information for the ESD item named on this record. The format of the behavioral attribute information is shown in Table 54 on page 212.
Name Length	70-71	Binary(2)	Length of the name of this ESD item. This length field cannot be zero.
Name	72-*	Byte(n)	Name of this ESD item. All names (including blank names) must be at least one character long. Trailing blanks should be removed.
Trailer		Byte(m)	Unused space at the end of a record is reserved and cannot be used for any other purpose.

External symbol definition continuation record

The format of an external symbol definition continuation record is as follows:

Table 51. External symbol definition continuation record

Field	Offset	Type	Description
PTV	0-2	Byte(3)	Type of record
			X'030200' If not continued
			X'030300' If continued
Text	3-*	Byte(n)	The next n bytes of the name.
Trailer		Byte(m)	Unused space at the end of a record is reserved, and cannot be used for any other purpose.

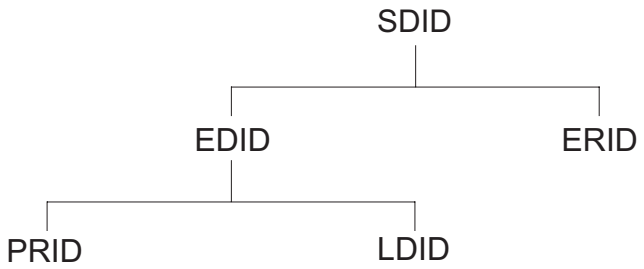
External symbol ID and name relationships

The following table summarizes the ESDIDs of items and their “Parent ESDIDs”. Terms like “SD ESDID” can be abbreviated as “SDID”, and so forth.

Table 52. Relationship of an element's ESDIDs and parent ESDIDs

If this ESD Item is:	Then its ESDID is:	And its Parent ESDID is:	And, the External Name defines a:
SD	SDID	zero	Section name
ED	EDID	SDID	Class name
LD	LDID	EDID	Label name
PR	PRID	EDID	Part name
ER	ERID	SDID	External name

These “parentage” relationships can also be viewed pictorially:



Note: ESDIDs must be defined before they appear in any referencing field.

Elements specifiable on ESD records

Table 53. Specifiable external symbol definition record items

Field	SD	ED	LD	PR	ER
Symbol Type	Y	Y	Y	Y	Y
ESDID	Y	Y	Y	Y	Y
Parent ESDID		Y	Y	Y	Y
Offset (see note)		A	Y		
Length		Y		Y	
Extended Attributes ESDID			Y		Y

Table 53. Specifiable external symbol definition record items (continued)

Field	SD	ED	LD	PR	ER
Extended Attribute Offset			Y		Y
Name Space ID (see note)		Y	Y	Y	Y
Fill Byte		Y			
Behavioral Attributes (see note)	Y	Y	Y	Y	Y
Symbol Name Length	Y	Y	Y	Y	Y
Symbol Name	Y	Y	Y	Y	Y

Note: "A" means "Currently, Assembler Only".

The Name Space assigned to a LD or PR item must match that of the ED to which it belongs.

See "ESD item behavioral attribute assignment" on page 216 for Behavioral Attribute assignments.

The owning parent ID of an SD must be zero.

External symbol definition behavioral attributes

The format of the behavioral attributes data field in an external symbol definition record is as follows:

Table 54. External symbol definition behavioral attributes

Field	Offset	Type	Description
Addressing Properties	0	Byte(1)	Addressing mode associated with this external symbol. If control is received from the operating system at an entry point named by this symbol, these addressing properties will determine the addressing mode. **X'00'** AMODE not specified (default=24) **X'01'** AMODE(24) **X'02'** AMODE(31) **X'03'** AMODE(ANY) (either 24-bit or 31-bit: an entry point can tolerate either addressing mode) **X'04'** AMODE(64) **X'10'** AMODE(MIN): the program management binder can set the AMODE to the minimum AMODE of all entry points in the program object. All other values are reserved.
Residence Properties	1	Byte(1)	Residence mode associated with this external symbol. **X'00'** RMODE not specified (default=24) **X'01'** RMODE(24) **X'03'** RMODE(31) **X'04'** RMODE(64) All other values are reserved. **Note:** RMODE(31) is equivalent to OBJ RMODE(ANY).

Table 54. External symbol definition behavioral attributes (continued)

Field	Offset	Type	Description
Text Record Style	2.0-3	Bit(4)	Designates the style of text to be accepted into this class. (See “Text record” on page 218 for further details.) **Note:** Text is valid only for ED and PR items. **B'0000'** Byte-oriented data. The “Address” field in the Text record provides the offset within the designated element where the data bytes are to be placed. **B'0001'** Structured-record (binder-defined) data. Only one form of structured-record data is currently defined (see “Identification record data field” on page 220 and Figure 22 on page 221 for a description). **B'0010'** Unstructured-record (user-defined) data All other values are reserved. **Note:** The Text Record Style in the ED record will be matched against that on the text records for this element, as a safety check. All text data within a class must be of the same style.
Binding Algorithm	2.4-7	Bit(4)	Type of binding action to be performed: **B'0000'** Concatenate: each of the section contributing elements within the designated class will be concatenated by placing the contributions “end to end”, with boundary alignment. (This is typified by current OBJ SD items.) **B'0001'** Merge: identically named parts will be “merged” by retaining the longest length and most restrictive alignment. (This is typified by current OBJ CM and PR items.) The merged parts will then be concatenated (with differently-named parts possibly having different alignments) in a program management binder-created class. Only parts are allowed in a MERGE class. All other values are reserved.

Table 54. External symbol definition behavioral attributes (continued)

Field	Offset	Type	Description
Tasking Behavior	3.0-2	Bit(3)	Translators wanting to specify the “traditional” tasking and concurrency attributes can use the following settings: **B'000'** Unspecified **B'001'** NON-REUS: Not serially reusable **B'010'** REUS: serially reusable **B'011'** RENT: reentrant All other values are reserved. Note that RENT implies REUS.
	3.3	Bit(1)	Reserved.
Read-Only	3.4	Bit(1)	Read-only indicator; no stores are allowed into this object, so the system can place it into protected storage. **B'0'** Not read-only **B'1'** Read-only
Executable	3.5-7	Bit(3)	Executable or not-executable indicator **B'000'** Not specified (currently) **B'001'** Not executable (That is, “This is data”.) **B'010'** Executable (That is, “This is code”.) All other values are reserved. **Note:** These flags can be applied to LD, PR, and ER elements. LD elements can also inherit the executability properties of the element to which they belong.
	4.0-4.1	Bit(4)	Reserved.
Duplicate symbol severity	4.2–3	Bit(2)	Severity to be associated with duplicate definitions of this symbol. **B'00'** Severity determined by the binder. **B'01'** Severity should be at least 4 (warning). **B'10'** Severity should be at least 8 (error). **B'11'** Reserved. **Note:** This field applies to PR ESD items only.

Table 54. External symbol definition behavioral attributes (continued)

Field	Offset	Type	Description
Binding Strength	4.4-7	Bit(4)	Strength of a definition or reference: **B'0000'** Strong reference or definition **B'0001'** Weak reference or definition All other values are reserved. Weak references are handled as in current products; the interactions with definitions are as follows: **Strong Reference** If unresolved, an "out-of-module" external library search will be made for a name to resolve the reference. **Weak Reference** If unresolved, no "out-of-module" external library search will be made to resolve the reference. **Strong Definition** Can be resolved to any reference. (This is the default, and normal, definition strength.)
Class Loading Behavior	5.0-1	Bit(2)	Determines whether or not the elements in this class will be loaded with the module when a LOAD (or similar) request to the operating system is satisfied by bringing the program object into storage. **B'00'** **LOAD**–Load this class with the module (Also known as INITIAL LOAD.) **B'01'** **DEFERRED LOAD**–This class will very probably be required by the program, and should be partially loaded in preparation for such a request. **B'10'** **NOLOAD**–Do not load this class with the module. **B'11'** Reserved.
COMMON Flag	5.2	Bit(1)	If 1, indicates that this section should be treated as an "old" COMMON: that is, as like any other CM section. If more than one COMMON is present, the longest length will be retained; if an SD section with the same name is present, its length and text will be retained. The only text class supported is B_TEXT.
Direct Versus Indirect Reference	5.3	Bit(1)	This bit indicates whether references to the symbol are direct or via a linkage descriptor. This bit is also known as the "descriptor bit". **B'0'** References are direct (as far as binder processing is concerned). **B'1'** For a PR item, indicates that the item represents a linkage descriptor. For An ER item, if the XPLINK bit is on, this represents a reference to an XPLINK linkage descriptor.

Table 54. External symbol definition behavioral attributes (continued)

Field	Offset	Type	Description
Binding Scope	5.4	Bit(4)	Requested binding or resolution-search scope of an external symbol. **B'0000'** Unspecified **B'0001'** Section scope (“local”) **B'0010'** Module scope (“global”) **B'0011'** Library scope. **B'0100'** Import-Export scope All other values are reserved.
	6.0-1	Bit(3)	Reserved.
Linkage Type	6.2	Bit(1)	Linkage Convention Indicator **B'0'** Standard OS linkage. This is the default. **B'1'** XPLINK linkage. **Note:** This bit is valid for ER, LD, PD, and PR items.
Alignment	6.3-7	Bit(5)	Storage alignment requirement of this object **Bits** **Implied Alignment of the Object** **B'00000'** byte **B'00001'** halfword **B'00010'** fullword **B'00011'** doubleword **B'00100'** quadword **B'01100'** 4KB page All other values are reserved. **Notes:** 1. Alignment bits = $\log_2$(boundary_size). 2. All these alignments are supported for ED records. All alignments except page alignment are supported for PR items.
	7-9	Byte(3)	Reserved.

ESD item behavioral attribute assignment

Table 55 shows which behavioral attributes apply to the different ESD items. Note that the attributes of PR items (with the exception of alignment) are determined from the element EDID to which they belong.

Table 55. Specifiable external symbol behavioral attributes

Field	SD	ED	LD	PR	ER
Addressing Properties			Y		Y
Residence Properties		Y			
Text Record Style		Y			
Binding Algorithm		Y			
Tasking Behavior	Y				

Table 55. Specifiable external symbol behavioral attributes (continued)

Field	SD	ED	LD	PR	ER
Read-Only		Y			
Executable		Y(1)	Y		Y
Binding Strength			Y		Y
Class Loading Behavior		Y			
COMMON Flag (2)	Y				
Alignment		Y		Y	

Notes:

1. The Executable property would apply to all LDs in this element.
2. The CM Flag may be set only for SD items with text class B_TEXT. It is invalid if the SD contains any other text class.

ESD Extended Attributes

In the ESD record for each external symbol, there is a pair of 4 byte fields located at offsets 28 and 32 for the Extended Attribute ESDID and offset, respectively. These two items locate the place in the object where the descriptive information about the symbol is found. **The ESDID identifies the element. The offset is calculated with respect to the beginning of the element identified by the ESDID.** The element must reside in the same section as the ESD record specifying the extended attribute information.

Extended Attribute information is represented using the following data structures:

Table 56. Extended attribute information data structures

Field	Length/Type	Meaning
Text length	Unsigned 32-bit integer	The length of the full attribute text including this field.
Extended attributes architecture level	Unsigned 16-bit integer	The level of definition for this extended attribute data structure.
Number of attribute entries	Unsigned 16-bit integer	The number of extended attribute data elements for this ESD or RLD item

Only architecture level 1 is currently supported.

Following the extended attributes header are one or more 8-byte data element descriptors. Each such descriptor is a structure of the following form:

Table 57. Data element descriptor data structures

Field	Length/Type	Meaning
Attribute type	Unsigned 16-bit integer	The type of attribute information.
Offset to data element	Unsigned 16-bit integer	The offset from the origin of the extended attributes header to the first byte of the data element associated with this descriptor.
Data element length	Unsigned 16-bit integer	The length in bytes of the data element associated with this descriptor.
Reserved	2 bytes	Reserved.

These data element descriptors are immediately followed by the data elements.

The following attribute codes are currently supported:

Table 58. Supported extended attribute type codes

Type Code	Attribute Assignment
0	Not defined; reserved.
1	XPLINK call descriptor for referenced function.
2–65535	Reserved.

IBM products currently place the attribute information in an initial load class named C_EXTNATTR.

Text record

This GOFF definition describes three basic mechanisms or *styles* of supplying text information:

- **Byte-oriented** data is an unstructured stream of bytes to be placed at a specified offset (“address”, for Assembler users) within a designated element of the program object.

  Byte-oriented data is typified by machine language instructions and data (as in current OBJ TXT records). (These can be thought of as analogous to “U-format” inputs.)
- **Structured-record** data is in the form of fixed-length records whose internal structure is known to and defined by the program management binder, and which will be placed in the `C'B_IDRL'` class. (These can be thought of as analogous to “F-format” inputs.)

  The only currently acceptable form is 19-byte IDR data. (See Figure 22 on page 221.) Data records of any other length or content must be defined by the program management binder.

  An example of structured-record data is current OBJ IDR data, which is in the form of one or two 19-byte structures on an OBJ END record.
- **Unstructured-record** data is in the form of records whose internal structure is (and will remain) unknown to the program management binder, which will simply append such records (prefixed by a length field) at the current end of the designated element. The length of each such record must be supplied by its provider. (These can be thought of as analogous to “V-format” inputs.)

  Examples of unstructured-record data are current OBJ SYM data and GOFF ADATA records.

A text record has the following format:

Table 59. Text record

Field	Offset	Type	Description	
PTV	0-2	Byte(3)	Type of record	
			X'031000'	If not continued
			X'031100'	If continued
	3.0-3	Bit(4)	Reserved.	

Table 59. Text record (continued)

Field	Offset	Type	Description
Text Record Style	3.4-7	Bit(4)	Designates the style of text data on this record: **B'0000'** Byte-oriented data. The "Address" field provides the position where the data bytes are to be placed. **B'0001'** Structured-record data **Note:** All structured-record data in a text record must have the same format. **B'0010'** Unstructured-record data All other values are reserved. **Note:** This field replicates a similar field in the element definition, so that text records can be checked for consistency.
Element ESDID	4-7	Binary(4)	ESDID of the **element** or **part** to which the data on this (and any subsequent continuation) records belongs. **Note:** Since the ED or PR record contains both the class name and the ESDID of the Section Definition (SD) record, this field uniquely identifies the program object element to which this data belongs.
	8-11	Binary(4)	Reserved.
Offset	12-15	Binary(4)	Starting offset from the element or part origin of the text on this (and any subsequent continuation) records. This field must be zero for structured-record and unstructured-record styles of data. **Note:** The offset for Assembler-produced text can be relative to a nonzero element origin.
Text Field True Length	16-19	Binary(4)	If the Text Encoding Type in the following field is zero, this field must be zero. If the Text Encoding Type is nonzero, this field specifies the length of the text after expansion.
Text Encoding	20-21	Binary(2)	If the text on this record is not compressed or not encoded, this field must be zero. A nonzero value indicates that the data is encoded, and will require decoding, expansion, or other treatment. See "Text encoding and compression" on page 222.
Data Length	22-23	Binary(2)	Total length in bytes of the data on this (and any following continuation) record. This length is unsigned and cannot be zero.
Data	24-*	Byte(n)	The next n bytes of data. **Note:** The IDR data must follow conventions and rules for translator-produced IDR data. (See Figure 22 on page 221.) The format for emitting IDR data is given in "Identification record data field" on page 220 below.
Trailer		Byte(m)	Unused space at the end of a record is reserved, and cannot be used for any other purpose.

Text continuation record

The format of a text continuation record is as follows:

Table 60. Text continuation record

Field	Offset	Type	Description
PTV	0-2	Byte(3)	Type of record **X'031200'** If not continued **X'031300'** If continued
Data	3-*	Byte(n)	Up to n bytes of data.
Trailer		Byte(m)	Unused space at the end of a record is reserved and cannot be used for any other purpose.

Identification record data field

Identification records have a fixed format and fixed content, both defined by the program management binder. Each Identification Data field of a (structured-record) data record has the following format:

Table 61. Identification record data field

Field	Offset	Type	Description
	0	Byte(1)	Reserved
Type	1	Byte(1)	Type of IDR item: **X'00'** Primary-translator identification data (Format 1) (see “Current (old) IDR data (format 1)” on page 221) **X'01'** Secondary-translator identification data (Format 1) **Note:** This corresponds to the second IDR field on current OBJ END record. **X'02'** Extended Identification Data (Format 2) (see “Extended IDR data (format 2)” on page 221) **X'03'** Primary-translator identification data (Format 3) (see “Extended IDR data (format 3)” on page 221) **X'04'** Secondary-translator identification data (Format 3) All other values are reserved.
Length	2-3	Binary(2)	Total length in bytes of the data in this field. This length cannot be zero.
IDR Data	4-*	Byte(n)	The n bytes of identification data **Note:** Multiple IDR data items are acceptable; it is up to the language translator producing the IDR data to supply more than one value, or to provide appropriate syntax permitting added IDR values to be expressed in its input language. All data is in the form of single-byte EBCDIC characters.

Remember that the IDR Data Field begins at byte offset +24 (the “Data” field) in the Text Record.

Current (old) IDR data (format 1)

The “old” format of IDR data is known in the program management binder as “IDRL” data, because it is typically placed either by default or explicitly into the B_IDRL class.

The 19-byte Format 1 IDR data entry is shown (using Assembler Language notation) in the following figure.

```
Translator   DS   CL10     Translator identification ("PID Number")
Version      DS   CL2      Version Level (01 to 99)
Release      DS   CL2      Release Level (01 to 99)
Trans_Date   DS   CL5      Translation date in Julian format (YYDDD)
```

Figure 22. Current IDR data item (format 1)

If the length of the Translator Identifier is less than 10 characters, it must be padded on the right with blanks.

Notes:

1. The binder treats 2-digit years in IDR data as follows:

```
if  ( YY > 65 )
    then  Year = 1900+YY
    else  Year = 2000+YY
```

2. IDR data is not stored in load modules by the Linkage Editor and program management binder in precisely this format.

Extended IDR data (format 2)

The “extended” format of IDR data is known in the program management binder as “IDRU” (“User IDR”) data, because it is placed into the B_IDRU class (for which an ED must have been previously defined).

The format of an extended IDR data entry is shown (using Assembler Language notation) in the following figure.

```
Date         DS   PL4      Packed Decimal Julian Date, as YYYYDDDF
Data_Length  DS   H        True length of following IDR data text
IDR_Data     DS   CL80     IDR data (true length in previous 2 bytes)
```

Figure 23. Extended IDR data item (format 2)

The IDR data can be up to 80 characters long. The IDR “record” must be 86 characters long, even if the IDR data text is less than 80 characters.

Note: The content of the 80-byte IDR_Data field has not been architected. (It is generated internally by the binder, not by any translator.)

Extended IDR data (format 3)

To accommodate year 2000 compatibility, a 30-byte Format 3 IDR data structure is provided to support four-digit year values and time stamps. It is treated as “IDRL” data.

```
Translator   DS   CL10     Translator identification ("PID Number")
Version      DS   CL2      Version Level (01 to 99)
Release      DS   CL2      Release Level (01 to 99)
Trans_Date   DS   CL7      Translation date in Julian format (YYYYDDD)
Trans_Time   DS   CL9      Translation time (HHMMSSTTT)
```

Figure 24. IDR data item (format 3)

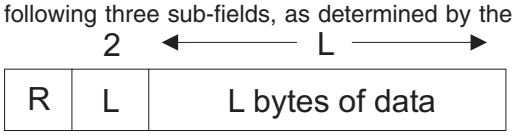
The only differences between Format 3 and Format 1 IDR data are that the `Trans_Date` field has been extended from five to seven characters, with `YYDDD` in Format 1 becoming `YYYYDDD` in Format 3, and the nine-character `Trans_Time` "time stamp" field has been added.

Text encoding and compression

In those cases where some value is gained by a translator's encoding, compressing, or encrypting some or all of the text, the following structures appear in the "Data" portion of the record described in Table 59 on page 218, depending on the "Text Encoding Type". Only one method of encoding is supported.

Note: Run-time decoding and/or decryption is the responsibility of the loaded program, which should place its encoded text into a special class that can be loaded (on demand) at run time.

Table 62. Text encoding types

Text Encoding	Data Field
X'0001'	The data field of the text record contains one or more instances of the following three sub-fields, as determined by the "Data Length" field:  • A nonzero unsigned repeat count R (Binary(2)) • A nonzero unsigned string length L (Binary(2)) • A string of bytes (Char(L)) whose length is specified by the preceding string length sub-field. The expanded text will be R×L bytes long and will contain R copies of the L bytes.
X'0002'- X'FFFF'	Reserved.

Relocation directory record

A relocation directory ("RLD") record has the following format:

Table 63. Relocation directory record

Field	Offset	Type	Description
PTV	0-2	Byte(3)	Type of record **X'032000'** If not continued **X'032100'** If continued
	3	Byte(1)	Reserved.
Length	4-5	Binary(2)	Total length of the relocation data on this record. This length cannot be zero.
Relocation Data	6-*	Byte(n)	Up to `n` bytes of relocation data. The format of the RLD data elements is shown in Table 64 on page 223.
Trailer		Byte(m)	Unused space at the end of a record is reserved and cannot be used for any other purpose.

Relocation directory data item

The format of a relocation directory ("RLD") data item is shown in Table 64 on page 223. Note that a relocation directory data item can be from 8 to 28 bytes long,

depending on which fields are present. The presence or absence of each field is determined by the “Flags” field. Offsets shown assume all fields are present.

Table 64. Relocation directory data element

Field	Offset	Type	Description
Flags	0-5	Byte(6)	Flags describing this RLD item. The flags are shown in Table 66 on page 224.
	6-7	Byte(2)	Reserved.
R Pointer	8-11	Binary(4)	ESDID of the ESD entry (ED or ER) which will be used as the basis for relocation. • For internal references, this will be the ED ESDID defining the referenced element. (The offset of the referenced position within the referenced element will have been placed by the translator in the target text field of the address constant.) • For external references, this will be the ER or PR ESDID describing the referenced symbol.
P Pointer	12-15	Binary(4)	ESDID of the element within which this address constant resides.
Offset	16-19	Binary(4)	Offset within the element described by the P pointer where the adcon is located; the place where the address constant can be found; the position of the field to be updated or relocated. This field is called the *fixup target*, *relocation target*, or simply *target*. **Note:** The constant part of a translator's address expression is stored in the “Target Field” (at this offset, in the element defined by the P pointer). It can contain a constant, an offset, or be ignored (“not fetched”).
	20-23	Binary(4)	Reserved.
	24-27	Binary(4)	Reserved.

Note that RLD records will normally be smaller if the RLD data is sorted by P-pointer, because typical text elements contain more than a single adcon.

Relocation directory continuation record

The format of a relocation directory continuation record is as follows:

Table 65. Relocation directory continuation record

Field	Offset	Type	Description
PTV	0-2	Byte(3)	Type of record **X'032200'** If not continued **X'032300'** If continued
Data	3-*	Byte(n)	Up to `n` bytes of data.
Trailer		Byte(m)	Unused space at the end of a record is reserved, and cannot be used for any other purpose.

Relocation directory data element flags field

The format of the 6-byte *Flags* field of a relocation directory (“RLD”) data element is as follows:

Table 66. Relocation directory data element flags field

Field	Offset	Type	Description
Same R-ID	0.0	Bit(1)	Indicates whether or not this RLD item has the same R pointer as the previous item. If yes, the R pointer (R-ID field) is omitted from this item. **B'0'** Different R pointer **B'1'** Same R pointer as the previous RLD item.
Same P-ID	0.1	Bit(1)	Indicates whether or not this RLD item has the same P pointer as the previous item. If yes, the P pointer (P-ID field) is omitted from this item. **B'0'** Different P pointer **B'1'** Same P pointer as the previous RLD item.
Same Offset	0.2	Bit(1)	Indicates whether or not this RLD element has the same offset as the previous item. If yes, the Offset field is omitted from this item. **B'0'** Different offset from the previous item. **B'1'** Same offset as the previous RLD item.
	0.3-4	Bit(2)	Reserved.
	0.5	Bit(1)	Reserved.
Offset Length	0.6	Bit(1)	Indicates the length of the Offset field in this RLD item: **B'0'** Offset is 4 bytes long.
Addressing Mode Sensitivity	0.7	Bit(1)	Indicates whether or not the final address should have its high-order bit set according to the addressing mode of its target. **B'0'** No addressing mode sensitivity **B'1'** Mode sensitive: set addressing-mode bit (or bits) in the address of this field, according to the AMODE of the R-pointer element. **Note:** This bit is set only for V-type constants. Regardless of this setting, specifying the HOBSET binder option sets the addressing mode bit in all V-type adcons which reference entry points whose addressing mode is 31 or any.
R-Pointer Indicators	1	Byte(1)	Indicates what type of data should be used as a “second operand” in this relocation or fixup action, and the type of referent. The two fields are shown below.

Table 66. Relocation directory data element flags field (continued)

Field	Offset	Type	Description
Reference Type	1.0-3	Bit(4)	Indicates what type of data will be used as the "second operand" of the relocation action: **0** R-address. **1** R-Offset relative to the start of the referent **2** R-Length. (The length value associated with an LD item is zero.) **6** R - Relative-Immediate (Instruction-Address-relative halfword or fullword offset, use with relative-immediate instructions). RI instruction allow one or more external symbols in their operands, however Binder only supports RI instructions with one external symbol in their operands. **7** R-type constant (The address of associated nonshared data area) **9** Long-Displacement 20-bit offset constant All other values are reserved.
Referent Type	1.4-7	Bit(4)	Indicates the type of item that is used as the referent of the relocation operation. For offset-type constants, indicates the type of item that is used as a base for the offset calculation **0** Label (R-ID restricted to LD) **1** Element (R-ID restricted to ED) **2** Class (R-ID restricted to ED). Note that classes do not have external names, so they have no associated ESDID. **3** Part (R-ID restricted to PR) All other values are reserved.
Action or Operation	2.0-6	Bit(7)	Indicates what type of operation should be performed using the "second operand" in this relocation or fixup action. (The first operand is the fetched value, or zero.) All operations are 32-bit (signed fullword integer) unless noted. In each case, the result of the operation replaces the first operand. **0** + (the second operand is added to the first) **1** – (the second operand is subtracted from the first) All other values are reserved.
Fixup Target Field Fetch/Store Flag	2.7	Bit(1)	Indicates whether or not the contents of the target field should be fetched and used in evaluating the following expression. • If this bit is zero, the contents of the fixup-target field will be "fetched" and used as the first operand in subsequent operations. • If this bit is one, the initial contents of the target field will be ignored, and the "fixup" quantity will be stored over it. If no initial contents has been fetched, the first operand's value is implicitly zero for the indicated operations.
	3	Byte(1)	Reserved.
Target Field Byte Length	4	Binary(1)	Unsigned byte length of the target field (adcon).

Table 66. Relocation directory data element flags field (continued)

Field	Offset	Type	Description
Bit Length, Bit Offset, and Conditional Sequential Flag	5	Byte(1)	If the target field is not an integral set of aligned bytes, these fields indicate the remaining length of the field in bits (i.e., the total bit length (mod 8)) and the offset of the first bit in the field. **Note:** The binder only supports when the target field byte length is 2 and the bit length is 4. This requirement is needed to support long-displacement 20 bit offset constant.
Bit Length	5.0-2	Bit(3)	If the target field is not an integral set of aligned bytes, this field and the Byte Length field provide the true length (in bits): Target_Field_Bit_Width = 8 × Byte_Length + Bit_Length.
Conditional Sequential Resolution Flag	5.3	Bit(1)	If 1, this bit means that the RLD item is part of a group of conditional sequential resolution address constants. That is, there is a following RLD item with the same P-ID, offset, and length but with a different R-ID. The final RLD item in the group must have this bit set to 0. The binder will attempt resolution of the first RLD in the group; if unable, it will continue with each sequential RLD item in the group until either a resolution is completed or none is possible. **Note:** Creators of this RLD item are urged to set the flag bits indicating *Same P-ID* and *Same Offset* to meet these conditions, in order to minimize the size of the RLD data for conditional sequential address constant groups. The binder requires that these additional conditions are met for all the RLD items that are part of the group. Thus all the RLD items in the group are equivalent to V-type address constants all referring to the same target field. • The Target Field Byte Length (at offset 4) must be identical. • The Reference Type (at offset 1.0-3) must be set to 0. • The Referent Type (at offset 1.4-7) must be set to 0. • The Fixup Target Field Fetch/Store Flag (at offset 2.7) must be set to 1.
	5.4	Bit(1)	Reserved
Bit Offset	5.5-7	Bit(3)	If the target field is not an integral set of aligned bytes, this field indicates the position of the first bit in the byte addressed by the Pointer and Offset that is a part of the field. **Note:** The binder only supports when the bit offset is 4. This requirement is used to support long-displacement 20 bit offset constant.

The relationships between the Reference Type and Referent Type are shown in the following table.

Table 67. RLD-element referent and reference types

Reference Type	Referent 0 (Label or Part)	Referent 1 (Element)	Referent 2 (Class)
Reference 0 (R-address)	A(label) or A(part)	Not supported	A(class origin) in which the label, element, or part belongs
Reference 1 (R-Offset)	Offset of label or part relative to class origin	Not supported	zero (offsets of classes are determined only at program loading time)
Reference 2 (R-Length)	Zero (labels don't have a length associated with them)	Length of the element or part	Length of the class to which the label, element, or part belongs

Examples of RLD data items

In this section, we will give some examples showing how RLD elements can be used.

- The four types of address constant used in current programs can be implemented very simply: the R- and P-pointers and P-position offset are set in the usual way, and the sequencing bits in the first byte of the Flags field are set according to whether or not the three values are repeated from the previous item. The Reference/Referent and Action bytes are set as follows:
  - A-con X'0000'
  - A-con (with subtraction of the R-value) X'0002'
  - V-con X'0001'
  - Q-con X'1200'
  - CXD-con X'2201'

  The Target Field Byte Length is set appropriately (for example, to 4, for 4-byte adcons), and the remaining fields are set to zero.

Deferred element length record

A deferred-length (“LEN”) record can be used to supply the true length of an element whose length was not known at the time the External Symbol Dictionary (ESD) record was issued. The format of a LEN record is as follows; a LEN record cannot be continued.

Table 68. Deferred section-length record

Field	Offset	Type	Description
PTV	0-2	Byte(3)	Type of record (X'033000')
	3-5	Byte(3)	Reserved.
Length	6-7	Binary(2)	The length of the deferred-length data items that follow. This length cannot be zero.
Element-Length Data	8-*	Byte(n)	Up to n bytes of element-length data. (See Table 71 on page 229.)
Trailer		Byte(m)	Unused space at the end of a record is reserved, and cannot be used for any other purpose.

The format of each 12-byte Deferred Element Length Data Element is as follows:

Table 69. Deferred element-length data item

Field	Offset	Type	Description
ESDID	0-3	Binary(4)	ESDID of the element for which this length value is supplied.
	4-7	Binary(4)	Reserved.
Length	8-11	Binary(4)	Length of the element.

A deferred element length data item cannot be split across records.

Associated data (ADATA) record

Associated data records can be created by a translator for direct inclusion with the program object as follows:

- Issue an element definition (ED) record for the class into which the records are to be placed. The class name is chosen as shown in "Associated data (ADATA) record types." Retain the ED associated with that type of ADATA record for use as each is emitted.
- In the Behavioral Attributes field, set all fields to zero except:
  - Class Loading Behavior is set to `B'10'`.
  - Binding Algorithm is set to `B'0000'` (CAT).
  - Set the "Text Record Style" to `B'0010'`, indicating that the data in this class is "unstructured" (so that the program management binder will not attempt to interpret it in any way).
  - Two other fields could be set: Read-Only and (not) Executable.
- For each ADATA record, create a GOFF text record as follows:
  - Set the Element ESDID to the ED value created for the class to which this type of record belongs.
  - Set the Offset to zero.
  - Set the Data Length to the length of the ADATA record, and fill the DATA field with the actual ADATA record itself.

The program management binder will accumulate the records of each type into the specified class, by appending each new record to the end of the previous record in that class.

Associated data (ADATA) record types

The following assignments have been made for ADATA record types:

Table 70. Associated data (ADATA) record type assignments

Type	Description
X'0000-7FFF'	Translator (and other components) records. (Note that some values in the range X'0000-0130' are already in use by High Level Assembler, COBOL/370, and OS/2 PL/I.)
X'8000-8FFF'	Program Management records.
X'9000-DFFF'	Reserved.
X'E000-EFFF'	Reserved for non-IBM translators.
X'F000-FFFF'	User records. IBM products and non-IBM translators will not create records with types in this range.

Class names are assigned to ADATA records in a very simple way: take the four hexadecimal digits of the ADATA record type, convert them to character format, and append them to the stem `C'C_ADATA'`. For example, the Class name for an ADATA record of type X'0000' would be `C'C_ADATA0000'`.

End of module record

The format of a module end ("END") record with an optional entry point request is as follows:

Table 71. End-of-module record, with optional entry point request

Field	Offset	Type	Description
PTV	0-2	Byte(3)	Type of record **X'034000'** If not continued **X'034100'** If continued
	3.0-5	Bit(6)	Reserved.
Flags	3.6-7	Bit(2)	Indicator flags, indicating presence or absence of a requested entry point: **B'00'** No entry point is suggested or requested on this END record. No subsequent fields on this record (other than the Record Count) are valid, and they cannot be used for any purpose. **B'01'** An entry point is requested by internal offset and ESDID on this END record. This ESDID can be either an EDID if the nominated entry point is known to be within this entry point. or an ERID if the nominated entry point is at an external symbol referenced in this input module. The Name Length field is zero. No continuation records are allowed. **B'10'** An entry point is requested by external name on this END record. The ESDID and Offset fields must be zero. Continuation records are allowed. Note that if an entry point is requested, the entry point name or class must be in namespace 1. **B'11'** Reserved.
AMODE	4	Byte(1)	AMODE of requested entry point. See AMODE values in "External symbol definition behavioral attributes" on page 212.
	5-7	Byte(3)	Reserved.
Record Count	8-11	Binary(4)	Count of GOFF logical records in this object file, including HDR and END records. If no record count is provided, this field must contain binary zero.
ESDID	12-15	Binary(4)	ESDID of the element containing the requested module internal entry point. If no entry point is requested, or if the entry point is requested by name, this field must be zero. If an ESDID is specified, the following Offset field is used.
	16-19	Binary(4)	Reserved.

Table 71. End-of-module record, with optional entry point request (continued)

Field	Offset	Type	Description
Offset	20-23	Binary(4)	Translator-assigned offset (or address, for Assembler output) of the requested module internal entry point. If no entry point is requested, or if the entry point is requested by name, this field must be zero. **Note:** Offset cannot be specified for external (ERID) entry point nominations.
Name Length	24-25	Binary(2)	Total length of the external name. This length cannot be zero if a name is present. It must be zero if an entry point is requested by ESDID and Offset, or if no entry point is requested.
Entry Name	26-*	Byte(n)	The first n characters of the external name requested as the module entry point.
Trailer		Byte(m)	Unused space at the end of a record is reserved, and cannot be used for any other purpose.

End of module continuation record, with optional entry point name

The format of an end-of-module continuation record is as follows:

Table 72. End-of-module (entry point name) continuation record

Field	Offset	Type	Description
PTV	0-2	Byte(3)	Type of record **X'034200'** If not continued **X'034300'** If continued
Entry Name	3-*	Byte(n)	Up to n bytes of data.
Trailer		Byte(m)	Unused space at the end of this record is reserved, and cannot be used for any other purpose.

Mapping object formats to GOFF format

Migration of current OBJ usage to GOFF style is straightforward, and is necessary only if the new GOFF facilities are needed. (The program management binder will continue to accept OBJ records.)

In general, the first step in creating a GOFF record will be to initialize the record buffer to binary zeros; most of the fields will not be needed, and zeros will provide the proper default values.

The GOFF analogs of current OBJ facilities are shown here, along with examples of the kinds of Assembler Language statements or operands that create them:

- ESD records

  OBJ ESD records can contain the following types of information:

  – SD names, including PC elements (blank SD names): these are derived from START, CSECT, and RSECT statements.
  – CM names: these are derived from COM statements.
  – LD names: these are derived from ENTRY statements.
  – ER names (possibly, WX): these are derived from EXTRN and WXTRN statements, and from the names in V-type address constants.

  - PR names: these are derived from DXD statements, and Q-type address constants containing DSECT names.
- TXT records

  Current OBJ TXT records contain simply an ESDID (of the section to which the text belongs), a length, and an address. The contents of TXT records are derived from the machine language instructions and data generated during the assembly process.
- RLD records

  Each item on OBJ RLD records is derived from an A-type, V-type, or Q-type address constant, or from a CXD statement.
- END records

  Current OBJ END records can contain an optional entry point request (either as an external name, or as an ESDID/offset combination to request entry within a module), plus zero to two IDR elements. The data on the END record is created by the translator (first IDR element), or is derived from the operand(s) of the END statement (requested entry point and second IDR item).
- SYM records

  Current OBJ SYM records have no analog in GOFF records; they are produced when the Assembler's TEST option is specified. The information currently collected on SYM records should be produced in GOFF ADATA records, or in some other translator-defined class. No mappings are shown for SYM data, nor for other forms of symbol tables.

It is assumed that translators will not produce nonstandard forms of object module records (for example, zero-data ESD records or blank SD items).

Module header records

The first record of a GOFF file must be a Module Header Record. It need only contain the PTV bytes X'03F000'.

Mapping object module ESD elements to GOFF format

This section describes how elements appearing in Object Modules are represented in the Generalized Object File Format.

Mapping object module ESD SD items

OBJ ESD SD elements require three GOFF ESD records: one to define the section, a second to define the class and element in which the text associated with the OBJ section is to reside, and the third to define an LD item for the external symbol named by the OBJ SD item. (GOFF SD names are not used for resolution; in effect, OBJ SDs are replaced by GOFF elementIDs.) Note that the LD item is not needed unless the OBJ SD name will be used to resolve external references.

These steps are shown in Table 73 on page 232.

Table 73. Mapping OBJ ESD SD items to GOFF format

OBJ Element	GOFF Element
SD element (1)	GOFF SD item: all fields are zero, except: • The Symbol Type is X'00' • The ESDID is that of the section • The Name Length is that of the OBJ SD name (cannot be zero), and the Name field contains the SD name. (Remember that blank names must contain a single blank.)
SD element (2)	The GOFF element (ED) record: all fields are zero, except: • The Symbol Type is X'01' • The ESDID is that of the element – Remember the element ESDID specified here, for use in subsequent TXT records belonging to this OBJ CSECT. • The Owning/Parent ESDID is that of the section just created • The Offset is zero (except possibly for Assembler output; the offset is then taken from the address field of the OBJ SD item) • The Length is that of the OBJ SD (or – if a deferred length is supplied later) • The Name Space ID is X'01' • The Behavioral Attributes are all zero (except for Residence Mode, if it was specified; for Assembler output, the Read-Only bit is set to the same value as the RSECT bit; and the Alignment is set to `B'0011'`) • The Name Length is set to 6, and the Name field is set to `C'B_TEXT'`.
SD element (3)	If the section name is blank, or is not used to resolve external references, this GOFF LD item record is not needed. Otherwise, all fields are zero, except: • The Symbol Type is X'02' • The ESDID is that of the LD name • The Owning ESDID is that of the element in which the text of the OBJ section is placed • The Name Space ID is X'01' • The Behavioral Attributes are set to zero (except that the Addressing properties are set to match those of the OBJ SD's AMODE) • The Name Length and Name are those of the OBJ SD element.

Mapping object module ESD PC items

PC items in OBJ format are actually defined by a blank SD name and a type of PC. However, PC items *could* have a nonblank name. Note also that an OBJ SD item with a blank name is invalid.

Table 74. OBJ treatment of ESD PC items and blank names

Name	PC	SD
blank	valid	invalid
nonblank	undefined	valid

The table shows that PC items are uniquely identified by a blank name, so that there is actually no need for a separate PC ESD type.

The same convention as for nonblank names applies for GOFF format; no special treatment is required for blank section names, except that no LD record (SD item (3)) is ever needed.

Thus, follow the same two steps as for SD items (1) and (2), using a Section Name of a single blank.

Mapping object module ESD LD items

OBJ ESD LD items behave in an identical fashion to those in GOFF records, and the mapping from OBJ to GOFF format is straightforward, as shown in Table 75.

Table 75. Mapping OBJ ESD LD items to GOFF format

OBJ Element	GOFF Element
LD element	The GOFF LD item record: all fields are zero, except: • The Symbol Type is X'02' • The ESDID is that of the LD name • The Owning ESDID is that of the element in which the text of the OBJ SD to which this LD belongs will be placed • The Offset is the offset of the LD within its OBJ SD section • The Name Space ID is X'01' • The Behavioral Attributes are set to zero (except that the Addressing properties can optionally be different from those of the OBJ SD's AMODE and Binding scope is set to B'0010'). • The Name Length and Name are those of the OBJ LD element

Mapping object module ESD ER/WX items

OBJ ESD ER and WX items are mapped into nearly identical GOFF forms, shown in Table 76.

Table 76. Mapping OBJ ESD ER/WX items to GOFF format

OBJ Element	GOFF Element
ER or WX	The GOFF ER record: all fields are zero, except: • The Symbol Type is set to X'04' • The ESDID is set to that of the ER item • The Owning ESDID is that of the section (SD) to which this ER belongs • The Behavioral Attributes are zero (except that the Binding scope field is set to B'0010' for WX items and B'0001' for ER items) • The Name Space ID is X'01' • The Name Length and Name are set appropriately

Mapping object module ESD CM items

OBJ ESD records containing CM data provide the CM name, its ESDID, and its length. (Alignment is implicitly doubleword.)

Note that Fortran BLOCK DATA sections can be handled as traditionally “overlaid” SD items if they are specified as follows:

OBJ CM items are mapped onto GOFF format SD, ED, and (if needed) LD records in *exactly* the same way as are SD parts (shown in “Mapping object module ESD SD items” on page 231), except that the “COMMON Flag” bit is set to `B'1'`.

Blank COMMON is treated specially by the binder (as it was by the linkage editor). Blank COMMON never has initializing text even if a blank-named control section (Private code) is present.

(The program management binder and the program object format support a much more powerful way to handle “COMMON” or external data previously emitted as OBJ SD items. They are then *not* mapped by the program management binder as

though they were SD items. This avoids the problem of order-dependent behavior in the OM/LM assignment of lengths to CM items with initializing data specified in an SD item.)

Mapping object module ESD PR items

OBJ ESD records containing PR data provide the PR name, its ESDID, and its length and alignment requirements.

OBJ PR elements are mapped onto GOFF format PR records as indicated in Table 77. A GOFF SD record defines the section owning the PR item; this record will have been previously issued. An ESD record defining an ED for class `C'B_PRV'` should also be issued, and the Binding Algorithm field is set to `B'0001'`.

Table 77. Mapping OBJ ESD PR items to GOFF format

OBJ element	GOFF element
PR element	The GOFF ESD record items are set as follows: all fields are zero, except: • The Symbol Type is set to X'03' (PR). • Its ESDID is assigned. • The owning ESDID is that of the owning ED (or zero if no ED record was provided, in which case the program management binder will take a default action). • The Length is set to that of the PR item. • The Name Space ID is X'02'. • The Behavioral Attributes are all set to zero, except that the Binding Algorithm field is set to `B'0001'` and the Alignment field is set to indicate the desired PR alignment. • The Name Length is set to that of the PR, and the PR name is placed in the Name field. Note that the use of “owning EDs” is not recommended, and translators producing GOFF records should not rely on the binder continuing to treat this as defaulting to B_PRV.

Mapping object module XSD items to GOFF format

The items on an XSD record map to GOFF ESD records in the same way as shown in “Mapping object module ESD SD items” on page 231, with the the following exception:

- Names need not be segmented

Other special flags in XSD records must be mapped to the appropriate fields in the GOFF ESD records; the details are not described here.

Mapping object module TXT items to GOFF format

OBJ text-record items are mapped onto GOFF format records as indicated in Table 78 on page 235. First, a GOFF record defining the element in which the text resides must have been issued, (see Table 73 on page 232 under “SD Item (2)”). Then, the GOFF text record would be created as follows; all other fields are zero.

Table 78. Mapping OBJ TXT items to GOFF format

OBJ Element	GOFF Element
TXT	Set “Text Record Style” to `B'0000'`.
Text ESDID	Element ESDID for the element in which the text resides. **Note:** This is the ESDID that was “remembered” when the element ED record was emitted. See Table 73 on page 232.
Offset	Offset
Text length	Data length

Mapping object module RLD items to GOFF format

GOFF RLD records are quite similar to OBJ RLD records; their function is the same, to describe address constants. The primary differences are that the “sequencing” bits refer to the *previous* RLD item rather than to the next, GOFF ESDIDs are 4 bytes long (rather than 2), the Offset field is 4 bytes (rather than 3), and the flags field is 6 bytes (rather than 1). The mappings are shown in Table 79. All fields should be initialized to zero.

Table 79. Mapping OBJ RLD items to GOFF format

OBJ Element	GOFF Element
Same R-, P- pointer Flags	Set bits as appropriate
R-, P- pointers and Offset	R, P pointers are expanded from 2 bytes to 4 (if present); Offset expanded from 3 bytes to 4. Note that in the OBJ implementation, the R-pointer for CXD items is set to zero. In GOFF files this should contain the EDID of the class whose length is requested. The binder treats zero R-ID as defaulting to B_PRV, but this should not be relied on.
R-Pointer Indicators and Action Bytes	The OBJ 4-bit type field is mapped to a 2-byte “Indicators and Action/Fetch” field: • `B'0000'` (A-type) becomes X'0000' • `B'0000'` (A-type with subtraction) becomes X'0002' • `B'0001'` (V-type) becomes X'0001' • `B'0010'` (Q-type) becomes X'1200' • `B'0011'` (CXD-type) becomes X'2201'
Target Field Byte Length	The OBJ 2-bit length field is mapped to the GOFF 8-bit Target Length field, and 1 is added.
Relocation Sign	This bit (X'02') is part of the “Action” field; see above.

Note: Incompatible or inconsistent OBJ uses (such as an A-con and a V-con referring to the same name) can be flagged by the program management binder.

Mapping object module END items to GOFF format

Mapping the information in OBJ END records will typically require two (and possibly three) GOFF records, the first for IDR information (shown in Table 80 on page 236), the second (possibly) if a deferred item length is desired (shown in Table 81 on page 236), and the third for END and (possibly) entry point information (shown in Table 82 on page 236). (The case where no deferred length is provided and no entry point is requested is shown in “Mapping object module END-entry items” on page 236.)

Mapping object module END IDR items

IDR data from OBJ END records is actually mapped into a text class named C'B_IDRL'. The procedure to follow is shown in Table 80.

Table 80. Mapping OBJ END IDR items to GOFF format

OBJ Element	GOFF Element
Initially:	Issue an ED record with class name C'B_IDRL', and set the Text Record Style to B'0001'. Assign the Owning ID of the section SD to which this element belongs. The EDID on this record will be used for the IDR data to follow (on one or more separate records).
IDR Data	Emit a GOFF Text record, with "Text Record Style" set to B'0001' (meaning "structured data"), ESDID set to the value defined on the ED record just described, Offset 0, and Data Length 23. The 23 bytes of data are: • 1 reserved byte • 1 byte set to X'00' (primary translator data) • 2 bytes containing decimal 19 (the length of the following data) • 19 bytes of normal IDR data.

Mapping object module END section-length elements

OBJ END records can supply the length of a section whose length was not known at the time the ESD SD item was produced. In the GOFF format, this information is provided on a new type of record ("LEN"), which must be emitted ahead of the END record. It is shown in Table 81. Note that the length must be that of an element, not of a program object section, which has no text or length associated with it. Note also that while OBJ END records can supply only a single deferred length (for a single SD item), GOFF records can accommodate multiple deferred lengths.

Table 81. Mapping OBJ END section–length items to GOFF format

OBJ Element	GOFF Element
END with section length	Emit a deferred-length record with Length 12, and data as follows: • 4-byte EDID for the element whose length is being provided. • 4 byte reserved field, set to zero. • 4-byte length for the specified element.

Mapping object module END-entry items

OBJ END records can request an entry point either by name or by ESDID and offset; these two forms are provided by the GOFF format.

Table 82. Mapping OBJ END–entry items to GOFF format

OBJ Item	GOFF Element
END (no entry point request)	Set the Flags bits to B'00', and the rest to zero. (In other words, nothing to do.)
END with entry point ED and offset	• Set the Flags bits to B'01', and the rest to zero. • Set the ESDID field to the EDID of the element in which the desired entry point is located. • Set the Offset to the entry point offset within the designated element.
END with entry point name	• Set the Flags bits to B'10'. • Set the 2-byte Name Length to the name of the requested entry point. Note that this name should already have appeared on an LD item (for an internal entry point) or an ER item (for an external entry point) in this object module.

Generating DLLs and linkage descriptors from GOFF object modules

Certain features in GOFF object modules must be used to build DLLs and DLL applications. A related function is the use of certain GOFF features to request the binder to generate linkage descriptors in a non-DLL situation. This support is primarily intended for Language Environment-conforming applications. In some cases Language Environment runtime support may have more stringent requirements than those enforced by the binder.

Exports

Exporting data

Exported data is represented as a PR or LD type ESD record with the binding scope set to Import-Export. IBM recommends using a PR type ESD item for compatibility with Language Environment. If using a PR ESD record, ensure the record length is greater than zero and set the name space ID to 3. If using an LD ESD record, you must set the 'executable' bit to 'not executable'. If you intend to reference the data in 64 bit addressing mode, set the addressing mode to AMODE(64). The 'linkage type' field must also match that of the referencing program.

Exporting code

Exported code is represented as a PR or LD type ESD record with the binding scope set to Import-Export. IBM recommends using an LD type ESD item for compatibility with Language Environment. When using a PR record, you must set the 'executable' bit to 'executable' and the name space ID to 3. The addressing mode and linkage type fields must match that of the referencing program.

Imports

Importing data

Reentrant data items are represented by a zero-length PR ESD record with the binding scope set to Import-Export. Non-reentrant data items (supported for XPLINK only) are represented by an ER ESD record with the binding scope set to Import-Export and the 'executable' bit set to 'non-executable'. The addressing mode and linkage type fields should match those of the exported data you wish to use.

The binder builds data descriptors and import information only if the ESD record for the imported data item is the target (R Pointer) of at least one RLD entry.

Non-XPLINK: When using reentrant code the owning ESD of the imported data item must be an ED for class C_WSA or C_WSA64. The binder builds a data descriptor for the reference in the owning class. Non-reentrant imported data is not supported by the binder. RLDs which reference non-XPLINK imported data must be Q-cons (RLD reference type=1).

XPLINK: The PR or ER in the importing class which represents the imported data must have the 'linkage type' field set to XPLINK. References to XPLINK data items should be A-type or V-type address constants (RLD reference type=0) contained in C_WSA or C_WSA64 when using reentrant code. References to XPLINK data items may be contained in any class when using non-reentrant code.

Importing code

A reference to imported code always requires a linkage descriptor. When using non-XPLINK code, the binder builds the required linkage descriptors. When using

XPLINK code, either the binder or Language Environment can build the require linkage descriptors. Language translators (compiler or assembler) may pass XPLINK descriptors to the binder.

The binder constructs descriptors in one of three classes: C_WSA, C_WSA64, or B_DESCR. The binder creates descriptors in B_DESCR only in the XPLINK case.

Non-XPLINK: You must use an ER ESD with the 'Indirect Reference' bit set to trigger the building of a linkage descriptor for the target symbol. Set the 'executable' attribute to 'code' and the scope to Import-Export.

An RLD entry whose target (R Pointer) is an ESD with the 'Indirect Reference' bit set resolves to the linkage descriptor. An RLD entry whose target is an ER ESD without the 'Indirect Reference' bit set resolves directly to the referenced symbol.

XPLINK:

1. **XPLINK 'call-by-name' references**

   'Call-by-name' refers to the case in which the name of the function being called is known at compile time. XPLINK call-by-name descriptors are generated by language translators. The binder recognizes these descriptors when they conform to the following format:

   - An XPLINK 'call-by-name' descriptor consists of two RLDs that describe a pair of contiguous adcons. These RLDs reference two 8-byte adcons for 64-bit code or two 4-byte adcons for 31-bit code.
   - The resident class must be C_WSA64 for 64-bit or C_WSA for 31 bit.

   Table 83. XPLINK 'call-by'name' descriptor adcon offsets

Offset	Contents
0	R(referenced symbol) - RLD reference type 7
4 or 8	V(referenced symbol) - RLD reference type 1

   The ER ESD describing the referenced symbol must have a linkage type of 'XPLINK,' a scope of Import-Export, and not have the 'Indirect Reference' bit set.

   When the target function resolves, the 'R-con' (the first RLD in the descriptor) resolves to the target function's environment if the referenced function provides one. The environment is defined by the 'Associated Data ID' field in the LD ESD record that defines the referenced function.

2. **XPLINK 'call-by-pointer' references**

   You may use a function pointer which points to a function descriptor if the function to be called is not known at compile time. The requirements for having the binder or Language Environment build the function descriptor are the same as for non-XPLINK imported code with the exception that you must set the linkage type to XPLINK in the ER ESD for the imported symbol. In addition, you must set the addressing mode to the expected addressing mode of the caller and callee. The binder uses the addressing mode to determine the length of the adcons used when building a descriptor.

Using linkage descriptors in non-DLL applications

Linkage descriptors can be used for code references when the bind is not for a DLL or DLL application (that is, DYNAM=DLL is not specified). The requirements are the same as described for references to imported code with exception that a scope of Import-Export is not necessary.

Appendix D. Binder API buffer formats

This topic contains general-use programming interface and associated guidance information.

This topic describes the external formats of program object data that are used to describe, bind, load and execute program objects.

Program object data is used as follows:

- Using the binder API, you can read, write or modify class-oriented data in the program object. API calls GETC, GETD, GETE, GETN and PUTD either create or accept module data in the external format. For more information, see Chapter 1, “Using the binder application programming interfaces (APIs),” on page 1.
- With minor exceptions,the fast data API can be used to retrieve the same data that is returned from the GETC, GETD, GETE, or GETN API calls. For more information, see Chapter 4, “IEWBFDAT - Binder Fast data access API functions,” on page 109.

Regardless of what method you use to process the program object data, you can use the IEWBUFF macro to allocate, initialize, map and delete buffers for each class of interest. For more information on using IEWBUFF, see Chapter 2, “IEWBUFF - Binder API buffers interface assembler macro for generating and mapping data areas,” on page 13.

All binder buffer formats are described in this appendix.

New buffer versions were introduced in the following releases:

- Version 1 - DFSMS/MVS Release 1
- Version 2 - DFSMS/MVS Release 3
- Version 3 - DFSMS/MVS Release 4
- Version 4 - OS/390 Version 2 Release 10
- Version 5 - z/OS Version 1 Release 3. New PMAR buffer and additional ESD fields added
- Version 6 - z/OS Version 1 Release 5. New CUI and LIB buffers, and new field in the Bindr Name List
- Version 7 - z/OS Version 1 Release 10.

The binder API, including the fast data API, support all formats. If VERSION is not specified on the IEWBUFF macro, version 1 buffers will be generated.

The differences between version 1 buffers and the later versions are very significant, especially in the case of the ESD records. If you are migrating an application from version 1 to a higher version number, see “Migration to version 2 buffers” on page 261.

The remainder of this section contains the external buffer formats for each version. The version 2 and 3 buffer sets are each followed by a discussion of migration from earlier buffer formats.

Data buffers are similarly structured. Each consists of a 32-byte buffer header followed by one or more entries, the number of entries being determined by the

© Copyright IBM Corp. 1991, 2012

SIZE or BYTES specification on the IEWBUFF macro. Each of the buffer descriptions show the buffer header and one buffer entry. The following notes apply to all formats and versions:

- The information stored in the buffer header can be used by your program to process data in any class or buffer version but should not be modified.
- Names do not appear in the buffer entries; instead, the names are represented by name lengths and pointers that locate the name string elsewhere in addressable storage. In the following record layouts, the name is shown as a 6-byte name field followed by its length and pointer components. Note that the 6-byte name field is not generated by IEWBUFF.
- Fields shown as reserved should be set to zero in input.
- ESD and RLD code values are defined symbolically in SYS1.MACLIB members IEWBCES and IEWBCRL, respectively.

Table 84. API buffer formats. The VERSION indicates the first API version that supported the buffer type. Note that when using IEWBIND with VERSION=1, the CLASS= parameter can specify either the class names beginning with "B_" or the obsolete equivalents which begin with "@" in place of "B_".

CLASS	TYPE=	VERSION	DESCRIPTION
@ESD or B_ESD	ESD	1	External Symbol Dictionary
@IDRB or B_IDRB	IDRB	1	Binder Identification Record (IDR)
@IDRL or B_IDRL	IDRL	1	Language Processor Identification Data (IDR)
@IDRU or B_IDRU	IDRU	1	User Identification Data (IDR)
@IDRZ or B_IDRZ	IDRZ	1	AMASPZAP Identification Data (IDR)
@RLD or B_RLD	RLD	1	Relocation Directory Data
@SYM or B_SYM	SYM	1	Internal Symbol Table
@TEXT or B_TEXT	TEXT	1	Program Text (instructions and data)
(none)	NAME	1	Binder Name List
(none)	XTLST	1	Extent List
B_MAP	MAP	2	Module Map
B_PARTINIT	PINIT	3	Part Initializer
B_PMAR	PMAR	5	Program Management Attribute Record
B_CUI	CUI	6	Compile unit information
B_LIB	LIB	6	Library path

Attention: If the version is omitted on the IEWBUFF and IEWBIND macros, it defaults to version 1.

The following table provides pointers to the version-specific definitions of the buffers.

Table 85. API buffer definitions

TYPE=	V1	V2	V3	V4	V5	V6	V7
CUI						283	
ESD	243	254	269	269	278	278	278
IDRB	245	245	245	245	245	245	245

Table 85. API buffer definitions (continued)

TYPE=	V1	V2	V3	V4	V5	V6	V7
IDRL	246	246	246	246	246	246	246
IDRU	247	247	247	247	247	247	247
IDRZ	248	248	248	248	248	248	248
LIB						282	
MAP		260	260	260	260	260	260
NAME	251	259	259	276	276	285	288
PARTINIT			274	274	274	274	274
PMAR					277	277	277
RLD	249	257	272	272	272	272	272
SYM	250	250	250	250	250	250	250
TEXT	250	250	250	250	250	250	250
XTLST	251	251	251	251	251	251	251

Note: For more information, see “Accessing program object class information” on page 306.

Buffer header contents

There are two ways to initialize the buffer header:

- Use the INITBUF function on the IEWBUFF macro
- Use the information found in Table 86 and Table 87 on page 242 to code it.

For additional information about the IEWBUFF macro, see “Using the IEWBUFF macro” on page 13.

The following general format applies to all binder buffers.

Table 86. Buffer header format. Table shows the formatting information for the Buffer header.

Field Name	Field Type	Offset	Length	Description
buffer_id	char	0	8	buffer type
buffer_leng	binary	8	4	buffer length in bytes (includes header)
version	binary	12	1	buffer version
*	binary	13	3	reserved, must be zero
entry_leng	binary	16	4	length of one entry
count	binary	20	4	number of entries
*	binary	24	8	reserved, must be zero

The count field designates the maximum number of entries that will be returned. The buffer must be large enough to hold the maximum number of entries or the call will fail. Use the entry length shown in Table 87 to compute the buffer size.

The buffer type corresponds to what is coded as the TYPE keyword on the IEWBUFF macro. Use the following values when retrieving data for the indicated buffer type:

Table 87. Buffer type format. Table shows the format information for the buffer type.

Buffer Type	buffer_id	entry_leng
CUI	IEWBCUI	80
ESD	IEWBESD	version 1 - 56 version 2 - 80 version 3 and later - 96
LIB	IEWBLIB	1
RLD	IEWBRLD	version 1 - 32 version 2 - 44 version 3 and later - 52
IDRU	IEWBIDU	96
IDRL	IEWBIDL	28
IDRZ	IEWBIDZ	24
IDRB	IEWBIDB	116
SYM	IEWBSYM	96
TEXT	IEWBTXT	1
NAME	IEWBBNL	version 1-6 - 28 version 7 and later - 36
XTLST	IEWBXTL	8
MAP	IEWBMAP	32
PINIT	IEWBPTI	1
PMAR	IEWBPMR	1
CUI	IEWBCUI	80

Version 1 buffer formats

Version 1 buffers are those supported in DFSMS/MVS version 1, releases 1.0 and 2.0, program management (PM1). They are also generated in later releases of DFSMS if VERSION=1 is specified or defaulted on the IEWBUFF macro. This section contains buffer layouts for each data class supported in PM1, as shown in Table 84 on page 240.

ESD entry (version 1)

```
     Field             Field  Off Leng          Description
     Name              Type   set

   IEWBESD                                 Binder ESD buffer, Version 1
   ESDH_BUFFER_ID      Char     0   8      Buffer identifier "IEWBESD"
   ESDH_BUFFER_LENG    Binary   8   4      Length of the buffer,
                                             including the header
   ESDH_VERSION        Binary  12   1      Version identifier
   *** RESERVED ***    Binary  13   3      Reserved, must be zeros
   ESDH_ENTRY_LENG     Binary  16   4      Length of each entry
   ESDH_ENTRY_COUNT    Binary  20   4      Number of entries in the buffer
   *** RESERVED ***    Binary  24   8      Reserved, must be zeros
   ESDH_ENTRY_ORIGIN           32          First ESD entry

   ESD_ENTRY                               ESD entry
4  ESD_TYPE            Char     0   2      ESD Type
       C'  '                                 Null Entry
       C'SD'                                 Control Section or
                                             Private Code
                                               CSECT if ESD_SCOPE=M
                                               private if ESD_SCOPE=S
       C'CM'                                 Common (Named or unnamed)
       C'LD'                                 Label Definition
       C'ER'                                 External Reference
                                               strong if ESD_SCOPE=L
                                               weak   if ESD_SCOPE=M
       C'PR'                                 Pseudoregister
5      C'ST'                                 Segment Table (Overlay)
5      C'ET'                                 Entry Table   (Overlay)
5      C'DS'                                 Dummy Section
5      C'PD'                                 Pseudoregister Definition
   ESD_SCOPE           Char     2   1      Scope of Name
       ' '                                   Not applicable
       C'S'                                  Section (Types SD/private,ST,ET)
       C'M'                                  Module (Types SD/CSECT,LD,
                                               ER/weak,CM,PR,DS,PD)
       C'L'                                  Library (Type ER/strong)
   ESD_TEXT            Char     3   1      Label Type
       C'I'                                  Instructions (Types SD,ET,ER,LD)
       C'D'                                  Data (Type CM,DS,ER,LD,PD,PR,ST)
       C' '                                  Unspecified
   ESD_ALIGN           Binary   4   1      Alignment Specification from
                                              language processor (Type PR)
       X'00'                                 Byte
       X'01'                                 Halfword
       X'02'                                 Fullword
       X'03'                                 Doubleword
   ESD_STORAGE         Binary   5   1      Storage Specification (Type SD)
       X'00'                                 Any Storage
       X'10'                                 Read-only Storage (Type SD)
   *** RESERVED ***    Binary   6   2      Reserved (must be zero)
   ESD_USABILITY       Binary   8   1      Reusability (Type SD)
       X'00'                                 Unspecified
       X'01'                                 Nonreusable
       X'02'                                 Reusable
       X'03'                                 Reentrant
       X'04'                                 Refreshable
```

Figure 25. Format for ESD entries (Part 1 of 2)

```
    Field              Field  Off Leng          Description
    Name               Type   set

   ESD_RMODE           Binary   9   1   Residence Mode (Type SD)
       X'00'                               Unspecified
       X'01'                               RMODE 24
       X'03'                               RMODE Any (24 or 31)
   ESD_AMODE           Binary  10   1   Addressing Mode (Type SD)
       X'00'                               Unspecified
       X'01'                               AMODE 24
       X'02'                               AMODE 31
       X'03'                               AMODE Any (24 or 31)
       X'04'                               use minimum AMODE
   *** RESERVED ***    Binary  11   1   Reserved, must be zero
   ESD_AUTOCALL        Binary  12   1   Autocall Specification (Type ER)
       X'80'                               Nevercall
2  ESD_STATUS          Bit     13   1   Resolution Status (Types ER,PR)
       1... ....                           Symbol has been resolved
       .1.. ....                           Symbol processed by autocall
       ..xx xxxx                           Reserved
   *** RESERVED ***    Binary  14   2   Reserved, must be zero
2  ESD_REGION          Binary  16   2   Overlay Region Number
                                             (Types SD,CM,ET)
2  ESD_SEGMENT         Binary  18   2   Overlay Segment Number
                                             (Types SD,CM,ET)
   ESD_LENG            Binary  20   4   Length of the defined section
                                           or pseudoregister
                                            (Types CM,SD,DS,ET,PD,PR,ST)
3  ESD_SECTION_OFFSET  Binary  24   4   Text offset within section
                                             (Types LD,PD)
2  ESD_MODULE_OFFSET   Binary  28   4   Text offset within module
                                             (Types SD,CM,LD,ST,ET,DS)
   *** RESERVED ***    Binary  32   2   Reserved (must be zero)
   ESD_NAME            Name    34   6   Symbol name. (Blank OK for
                                        private code and common.)
     ESD_NAME_CHARS    Binary  34   2     length of name in bytes
     ESD_NAME_PTR      Pointer 36   4     pointer to name string
   *** RESERVED ***    Binary  40   2   Reserved (must be zero)
2  ESD_TARGET          Name    42   6   Name of the section in which the
                                        symbol is defined (Type ER)
     ESD_TARGET_CHARS  Binary  42   2     length of name in bytes
     ESD_TARGET_PTR    Pointer 44   4     pointer to name string
   *** RESERVED ***    Binary  48   2   Reserved (must be zero)
1  ESD_RESIDENT        Name    50   6   Name of the section in which
                                        this ESD entry resides.
                                        (Types LD,ER,PR,PD)
     ESD_RESIDENT_CHARS Binary 50   2     length of name in bytes
     ESD_RESIDENT_PTR  Pointer 52   4     pointer to name string
```

Notes:

1. Ignored on input to the binder.
2. Recalculated by the binder.
3. Calculated on the ED and ER records, required input to LD.
4. ESD_TYPE is further qualified by ESD_SCOPE.
5. Binder-generated ESD type.

Figure 25. Format for ESD entries (Part 2 of 2)

Binder identification data (version 1)

```
     Field              Field  Off Leng        Description
     Name               Type   set

   IEWBIDB                                  Binder IDR Data, Version 1
   IDBH_BUFFER_ID       Char     0   8      Buffer identifier "IEWBIDB"
   IDBH_BUFFER_LENG     Binary   8   4      Length of the buffer,
                                              including the header
   IDBH_VERSION         Binary  12   1      Version identifier
   *** RESERVED ***     Binary  13   3      Reserved, must be zeros
   IDBH_ENTRY_LENG      Binary  16   4      Length of each entry
   IDBH_ENTRY_COUNT     Binary  20   4      Number of entries in the buffer
   *** RESERVED ***     Binary  24   8      Reserved, must be zeros
   IDBH_ENTRY_ORIGIN            32          First IDR entry


1  IDB_ENTRY                                Binder IDR Entry
   IDB_BINDER_ID        Char     0  10      Binder identification
   IDB_VERSION          Char    10   2      Binder version
   IDB_RELEASE          Char    12   2      Binder release
   IDB_DATE_BOUND       Char    14   7      Date of binding (YYYYDDD)
   IDB_TIME_BOUND       Char    21   6      Time of binding (HHMMSS)
   *** RESERVED ***     Binary  27   1      Reserved, must be zeros
   IDB_MODULE_SIZE      Binary  28   4      Length of module text
   IDB_CALLERID_CHARS   Binary  32   2      Number of significant characters
                                              in IDB_CALLERID
   IDB_CALLERID         Char    34  80      Caller identification
   *** RESERVED ***     Binary 114   2      Reserved, must be zeros
```

Notes:

1. Generated by binder during binding.

Figure 26. Format for binder identification data

Language processor identification data (version 1)

```
    Field              Field  Off Leng        Description
    Name               Type   set

   IEWBIDL                                  Language Processor IDR Record,
                                            Version 1
   IDLH_BUFFER_ID      Char     0   8       Buffer identifier "IEWBIDL"
   IDLH_BUFFER_LENG    Binary   8   4       Length of the buffer,
                                               including the header
   IDLH_VERSION        Binary  12   1       Version identifier
   *** RESERVED ***    Binary  13   3       Reserved, must be zeros
   IDLH_ENTRY_LENG     Binary  16   4       Length of each entry
   IDLH_ENTRY_COUNT    Binary  20   4       Number of entries in the buffer
   *** RESERVED ***    Binary  24   8       Reserved, must be zeros
   IDLH_ENTRY_ORIGIN           32           First IDR entry

3  IDL_ENTRY                                Language IDR Entry
   IDL_PID_ID          Char     0  10       Translator product ID
   IDL_VERSION         Char    10   2       Version of the translator
   IDL_MOD_LEVEL       Char    12   2       Modification level of translator
2  IDL_DATE_PROCESSED  Char    14   7       Date compiled or assembled
                                              (yyyyddd)
   *** RESERVED ***    Binary  21   1       Reserved, must be zeros
1  IDL_RESIDENT        Name    22   6       The name of the section to
                                            which this IDR data applies.
     IDL_RESIDENT_CHARS Binary 22   2         length of name in bytes
     IDL_RESIDENT_PTR  Pointer 24   4         pointer to name string
```

Notes:

1. Ignored on input to the binder.
2. If the module is saved as a load module, dates are truncated to 5 characters (YYDDD).
3. There normally is only one IDRL entry per section. However, if the section was produced by a multistage compilation, there is one IDRL record for each processor involved in the generation of the object code.

Figure 27. Format for language processor identification data

User identification data (version 1)

```
    Field              Field  Off Leng        Description
    Name               Type   set

   IEWBIDU                                  User IDR Data, Version 1
   IDUH_BUFFER_ID      Char     0   8       Buffer identifier "IEWBIDU"
   IDUH_BUFFER_LENG    Binary   8   4       Length of the buffer,
                                               including the header
   IDUH_VERSION        Binary  12   1       Version identifier
   *** RESERVED ***    Binary  13   3       Reserved, must be zeros
   IDUH_ENTRY_LENG     Binary  16   4       Length of each entry
   IDUH_ENTRY_COUNT    Binary  20   4       Number of entries in the buffer
   *** RESERVED ***    Binary  24   8       Reserved, must be zeros
   IDUH_ENTRY_ORIGIN           32           First IDR entry


   IDU_ENTRY                                User IDR entry
2  IDU_CREATE_DATE     Char     0   7       Date the entry was created
                                               (yyyyddd)
   *** RESERVED ***    Binary   7   1       Reserved, must be zeros
   IDU_DATA_CHARS      Binary   8   2       Number of significant characters
                                               in IDU_DATA
2  IDU_DATA            Char    10  80       Defined by the user
1  IDU_RESIDENT        Name    90   6       Name of section to which this
                                               IDR data applies
     IDU_RESIDENT_CHARS Binary 90   2         length of name in bytes
     IDU_RESIDENT_PTR  Pointer 92   4         pointer to name string
```

Notes:

1. Ignored on input to the binder.
2. If the module is saved as a load module, the identification data is truncated to 40 bytes and the creation date to 5 (yyddd).

Figure 28. Format for user identification data

AMASPZAP identification data (version 1)

```
    Field              Field  Off Leng        Description
    Name               Type   set

   IEWBIDZ                                  AMASPZAP IDR Data, Version 1
   IDZH_BUFFER_ID      Char     0   8   Buffer identifier "IEWBIDZ"
   IDZH_BUFFER_LENG    Binary   8   4   Length of the buffer,
                                           including the header
   IDZH_VERSION        Binary  12   1   Version identifier
   *** RESERVED ***    Binary  13   3   Reserved, must be zeros
   IDZH_ENTRY_LENG     Binary  16   4   Length of each entry
   IDZH_ENTRY_COUNT    Binary  20   4   Number of entries in the buffer
   *** RESERVED ***    Binary  24   8   Reserved, must be zeros
   IDZH_ENTRY_ORIGIN           32       First IDR entry


   IDZ_ENTRY                            Superzap IDR entry
2  IDZ_DATE            Char     0   7   Date of processing
                                        (yyyyddd)
   IDZ_ZAP_DATA        Char     7   8   PTF number or other ZAP data
   *** RESERVED ***    Binary  15   3   Reserved, must be zeros
1  IDZ_RESIDENT        Name    18   6   Name of the section to which
                                           this IDR data applies
     IDZ_RESIDENT_CHARS Binary 18   2     length of name in bytes
     IDZ_RESIDENT_PTR  Pointer 20   4     Pointer to name string
```

Notes:

1. Ignored on input to the binder.
2. If the module is saved as a load module, dates are truncated to 5 characters (YYDDD).

Figure 29. Format for AMASPZAP identification data

RLD entry (version 1)

```
    Field              Field  Off Leng        Description
    Name               Type   set

   IEWBRLD                                  Binder RLD buffer, Version 1
   RLDH_BUFFER_ID      Char     0   8   Buffer identifier "IEWBRLD"
   RLDH_BUFFER_LENG    Binary   8   4   Length of the buffer,
                                           including the header
   RLDH_VERSION        Binary  12   1   Version identifier
   *** RESERVED ***    Binary  13   3   Reserved, must be zeros
   RLDH_ENTRY_LENG     Binary  16   4   Length of each entry
   RLDH_ENTRY_COUNT    Binary  20   4   Number of entries in the buffer
   *** RESERVED ***    Binary  24   8   Reserved, must be zeros
   RLDH_ENTRY_ORIGIN           32       First RLD entry

   RLD_ENTRY                            RLD entry
   RLD_TYPE            Binary   0   1   Adcon Type
       X'10'                              Branch type (V-con)
       X'20'                              Nonbranch type (A-con)
       X'30'                              Pseudoregister (Q-con)
       X'40'                              Cumulative Pseudo-register Length (CXD)
       X'80'                              Long_Displacement 20-bit
                                            DL/DH offset (QY-con)
2  RLD_STATUS          Binary   1   1   Adcon Status
       X'01'                              References an unresolved symbol
       X'02'                              References a resolved symbol
       X'03'                              References a nonrelocatable
                                            symbol
2  RLD_ADCON_BDY       Binary  24   1   Binder no longer maintains
                                        Adcon boundary status and
                                        changes this value to 0
   RLD_ADCON_DIRECTION Binary   3   1   Adcon relocation direction
       X'00'                              Relocation is positive
       X'01'                              Relocation is negative
   RLD_ADCON_LENG      Binary   4   2   Length of the Adcon
   *** RESERVED ***    Binary   6   2   Reserved, must be zero
   RLD_SECTION_OFFSET  Binary   8   4   Offset of the address constant
                                           within the containing section
2  RLD_MODULE_OFFSET   Binary  12   4   Offset of the address constant
                                           within the module
   *** RESERVED ***    Binary  16   2   Reserved, must be zero
   RLD_TARGET          Name    18   6   Name of the external symbol to
                                           be used to compute the value
                                           of the adcon ("R-Pointer")
                                           adcon ("R Pointer")
     RLD_TARGET_CHARS  Binary  18   2     length of name in bytes
     RLD_TARGET_PTR    Pointer 20   4     Pointer to name string
   *** RESERVED ***    Binary  24   2   Reserved, must be zero
1  RLD_RESIDENT        Name    26   6   Name of the section to be used
                                           to compute the location of
                                           the adcon ("P-Pointer").
     RLD_RESIDENT_CHARS Binary 26   2     length of name in bytes
     RLD_RESIDENT_PTR  Pointer 28   4     Pointer to name string
```

Notes:

1. Ignored on input to the binder.
2. Recalculated by the binder.

Figure 30. Format for RLD entries

Internal symbol table (version 1)

```
    Field              Field  Off Leng        Description
    Name               Type   set

   IEWBSYM                                  Binder Symbol buffer, Version 1
   SYMH_BUFFER_ID      Char     0   8   Buffer identifier "IEWBSYM"
   SYMH_BUFFER_LENG    Binary   8   4   Length of the buffer,
                                           including the header
   SYMH_VERSION        Binary  12   1   Version identifier
   *** RESERVED ***    Binary  13   3   Reserved, must be zeros
   SYMH_ENTRY_LENGTH   Binary  16   4   Length of each entry
   SYMH_ENTRY_COUNT    Binary  20   4   Number of entries in the buffer
   *** RESERVED ***    Binary  24   8   Reserved, must be zeros
   SYMH_ENTRY_ORIGIN           32       First SYM entry

   SYM_ENTRY                            SYM entry
   SYM_CREATE_DATE     Char     0   7   Date entry created (yyyyddd)
   *** RESERVED ***    Binary   7   1   Reserved, must be zeros
   SYM_DATA_CHARS      Binary   8   2   Number of significant characters
                                           in SYM_DATA
   SYM_DATA            Char    10  80   Not defined by the binder
1  SYM_RESIDENT        Name    90   6   Name of the section to which
                                           the symbol data applies
     SYM_RESIDENT_CHARS Binary 90   2     length of name in bytes
     SYM_RESIDENT_PTR  Pointer 92   4     pointer to name string
```

Notes:

1. Ignored on input to the binder.

Figure 31. Format for symbol table (SYM) entries

Text data buffer (version 1)

```
    Field              Field  Off Leng        Description
    Name               Type   set

   IEWBTXT                              Binder Text buffer, Version 1
   TXTH_BUFFER_ID      Char     0   8   Buffer identifier "IEWBTXT"
   TXTH_BUFFER_LENG    Binary   8   4   Length of the buffer,
                                           including the header
   TXTH_VERSION        Binary  12   1   Version identifier
   *** RESERVED ***    Binary  13   3   Reserved, must be zeros
   TXTH_ENTRY_LENG     Binary  16   4   Length of entries (always 1)
   TXTH_ENTRY_COUNT    Binary  20   4   Number of entries (bytes) in the
                                           buffer
   *** RESERVED ***    Binary  24   8   Reserved, initialize to zeros
   TXT_ARRAY           Undef.  32  var  Program Text (length varies from
                                           1 to 2**31-1 bytes, depending
                                            on value in TXTH_ENTRY_COUNT)
```

Figure 32. Format for TXT entries

Binder name list (version 1)

```
    Field             Field  Off Leng          Description
    Name              Type   set

   IEWBBNL                              Binder Name List buffer, Ver 1
   BNLH_BUFFER_ID     Char     0   8    Buffer identifier "IEWBBNL"
   BNLH_BUFFER_LENG   Binary   8   4    Length of the buffer,
                                           including the header
   BNLH_VERSION       Binary  12   1    Version identifier
   *** RESERVED ***   Binary  13   3    Reserved, must be zeros
   BNLH_ENTRY_LENG    Binary  16   4    Length of each entry in the list
   BNLH_ENTRY_COUNT   Binary  20   4    Number of entries in the buffer
   *** RESERVED ***   Binary  24   8    Reserved, must be zeros
   BNLH_ENTRY_ORIGIN          32        First namelist entry

1  BNL_ENTRY                            Namelist entry
   BNL_NAME_CHARS     Binary   0   2    Number of significant characters
                                           in the class or section name
   *** RESERVED ***   Binary   2   2    Reserved, must be zeros
   BNL_NAME_PTR       Pointer  4   4    Address of the class or section name
```

Notes:
1. Output only.

Figure 33. Format for binder name list entries

Extent list (version 1)

```
    Field             Field  Off Leng          Description
    Name              Type   set

   IEWBXTL                              Module Extent List, Version 1
   XTLH_BUFFER_ID     Char     0   8    Buffer identifier "IEWBXTL"
   XTLH_BUFFER_LENG   Binary   8   4    Length of the buffer,
                                           including the header
   XTLH_VERSION       Binary  12   1    Version identifier
   *** RESERVED ***   Binary  13   3    Reserved, must be zeros
   XTLH_ENTRY_LENG    Binary  16   4    Length of each entry
   XTLH_ENTRY_COUNT   Binary  20   4    Number of entries in the list
   *** RESERVED ***   Binary  24   8    Reserved, must be zeros
   XTLH_ENTRY_ORIGIN          32        First extent list entry

1  XTL_ENTRY                            Extent List Entry
   XTL_LOAD_POINT     Binary   0   4    Load point address for extent
2  XTL_EXTENT_LENG    Binary   4   4    Length of extent
```

Notes:
1. Valid for output only. There is normally only one entry. When the binder is invoked under TSO, however, a second entry is provided for the mini-CESD.
2. The extent length includes the length of the module text plus other binder-related data.

Figure 34. Format for contents extent list entries

Version 2 buffer formats

VERSION=2 must be specified on the IEWBUFF macro to obtain buffers in this format.

Most buffers have not changed except for the version number, but VERSION 2 is supported by IEWBUFF for all classes, whether or not there is a format change. Differences between Versions 1 and 2 include:

- ESD and RLD buffers have changed significantly in support of multiple text classes, a new binding algorithm and new address constants.
- A new binder-defined class, B_MAP, describes the logical structure of the program object.
- External names can be up to 1024 bytes in length.
- @class-type class names are not supported in version 2 calls and buffers.
- For compatibility with PM1, the PM2 IEWBUFF macro produces the Version 1 buffer formats, described previously, if VERSION=1 is specified or defaulted in the macro invocation.

This section contains the buffer layouts that have changed or have been added in version 2. As indicated in Table 85 on page 240, all other buffer formats are the same for version 2 as in version 1.

ESD entry (version 2)

Please note that the version 2 ESD buffer has been significantly changed from the Version 1 buffer.

```
 Field              Field  Off Leng          Description
 Name               Type   set

IEWBESD                                  Binder ESD buffer, Version 2
ESDH_BUFFER_ID      Char     0   8   Buffer identifier "IEWBESD"
ESDH_BUFFER_LENG    Binary   8   4   Length of the buffer,
                                        including the header
ESDH_VERSION        Binary  12   1   Version identifier (Constant 2)
*** RESERVED ***    Binary  13   3   Reserved, must be zeros
ESDH_ENTRY_LENG     Binary  16   4   Length of each entry
ESDH_ENTRY_COUNT    Binary  20   4   Number of entries in the buffer
*** RESERVED ***    Binary  24   8   Reserved
ESDH_ENTRY_ORIGIN           32       First ESD entry

ESD_ENTRY                                ESD entry
ESD_TYPE            Char     0   2   ESD Type
    C'  '                                Null Entry
    C'SD'                                Control Section
    C'LD'                                Label Definition
    C'ER'                                External Reference
    C'PR'                                Part Reference
    C'PD'                                Part Definition
    C'ED'                                Element Definition

ESD_TYPE_QUAL       Char     2   2   ESD Type Qualifier
    C' '                                 (no qualification)
    C'SD'                                Section Definition (SD)
    C'CM'                                Common (SD)
    C'ST'                                Segment Table (SD)
    C'ET'                                Entry Table (SD)
    C'PC'                                Unnamed Section (SD)
    C'PR'                                Part Reference (PR, PD)
    C'PD'                                Part Definition (PR, PD)
    C'ER'                                External Reference (ER)
    C'WX'                                Weak Reference (ER)

ESD_NAME_SPACE      Binary   4   1   Name Space for symbols
    X'00'                                Section names
    X'01'                                Labels and references (LD, ER)
    X'02'                                Pseudoregisters (PR, PD)
    X'03'                                Parts (PR, PD) or ED record for
                                                 merge class
    X'04'-x'07'                          Reserved
ESD_SCOPE           Char     5   1   Scope of Name
    ' '                                  Not applicable
    'S'                                  Section (Types SD/private,ST,ET)
    'M'                                  Module (Types SD/CSECT,LD,
                                                ER/weak,CM,PR,DS,PD)
    'L'                                  Library (Type ER/strong)
ESD_NAME            Name     6   6   Symbol represented by ESD record
  ESD_NAME_CHARS    Binary   6   2     length of name in bytes
  ESD_NAME_PTR      Pointer  8   4     pointer to name string
*** RESERVED ***    Char    12   2   Reserved
1 ESD_RES_SECTION   Name    14   6   Name of containing section
  ESD_RESIDENT_CHARS Binary 14   2     length of name in bytes
  ESD_RESIDENT_PTR  Pointer 16   4     pointer to name string
ESD_LENG            Binary  20   4   Length of defined element
                                        (ED, PD, PR)
ESD_ALIGN           Binary  24   1   Alignment specification from
                                     language processor.  Indicates
                                     Alignment of section contribution
                                     within class segment (ED, PD, PR)
    X'00'                              Byte alignment (PD, PR)
    X'01'                              Halfword (PD, PR)
    X'02'                              Fullword (PD, PR)
    X'03'                              Doubleword (PD, PR, ED)
    X'04'                              Quadword (PR, PD, ED)
    X'0C'                              4K page (ED)
```

Figure 35. Format for ESD entries (Part 1 of 3)

```
    Field              Field  Off Leng          Description
    Name               Type   set

   ESD_USABILITY       Binary  25   1   Reusability of Section (SD)
       X'00'                              Unspecified
       X'01'                              Nonreusable
       X'02'                              Reusable
       X'03'                              Reentrant
       X'04'                              Refreshable
   ESD_AMODE           Bit     26   1   Addressing Mode for Section or
                                        label (SD, LD)
       X'00'                              Unspecified
       X'01'                              AMODE 24
       X'02'                              AMODE 31
       X'03'                              AMODE ANY (24 or 31)
       X'04'                              AMODE MIN
   ESD_RMODE           Bit     27   1   Residence Mode for class element
                                          (ED)
       X'01'                              RMODE 24
       X'03'                              RMODE ANY (24 or 31)
   ESD_RECORD_FMT      Binary  28   2   Record format for class (ED)
       H'1'                               Byte stream
       H'>1'                              Fixed length records
    Field              Field  Off Leng          Description
    Name               Type   set

   ESD_LOAD_FLAGS      Bit     30   1   Load Attributes (ED)
       1... ....                          Read-only
       .1.. ....                          Do not load with module
       ..1. ....                          Moveable
       ...1 ....                          Shareable
       .... 1111                          ** Reserved **
   ESD_BIND_FLAGS      Bit     31   1   Bind Attributes
2      1... ....                          Binder generated (SD, ED, LD)
2      .1.. ....                          No class data available (ED)
2      ..1. ....                          Variable length records (ED)
       ...1 ....                          Descriptive data (not text) (ED)
       .... 1111                          ** Reserved **
   ESD_BIND_CNTL       Bit     32   1   Bind control information
1      1... ....                          Removable class(ED)
1      .x.. ....                          ** Reserved **
1      ..xx ....                          Binding method (ED)
1      ..00 ....                            CAT (Catenated text)
1      ..01 ....                            MRG (Merged parts)
1      ..1x ....                            ** Reserved **
   *** RESERVED ***    Char    33   1   Reserved
   ESD_XATTR_CLASS     Name    34   6   Extended attributes class
                                          (LD, ER)
     ESD_XATTR_CLASS_CHARS
                       Binary  34   2     length of name in bytes
     ESD_XATTR_CLASS_PTR
                       Pointer 36   4     pointer to name string
   ESD_XATTR_OFFSET    Binary  40   4   Extended attributes element offset
                                        (LD, ER)
```

Figure 35. Format for ESD entries (Part 2 of 3)

```
2  ESD_SEGMENT          Binary  44   2   Overlay segment number (SD)
2  ESD_REGION           Binary  46   2   Overlay region number (SD)
   ESD_SIGNATURE        Char    48   8   Interface signature
2  ESD_AUTOCALL         Binary  56   1   Autocall specification (ER)
       x... ....                           ** Reserved **
       .1.. ....                           Entry in LPA.  If ON, name is
                                           an alias.
       ..xx xxxx                           ** Reserved **
2  ESD_STATUS           Bit     57   1   Resolution status (ER)
       1... ....                           Symbol is resolved
       .1.. ....                           Processed by autocall
       ..xx xxxx                           Reserved
2  ESD_TGT_SECTION      Name    58   6   Target section (ER)
     ESD_TARGET_CHARS   Binary  58   2     length of name in bytes
     ESD_TARGET_PTR     Pointer 60   4     pointer to name string
   *** RESERVED ***     Char    64   2   Reserved
   ESD_RES_CLASS        Name    66   6   Name of containing class
                                           (LD, PD) or target class (ER)
     ESD_RES_CLASS_CHARS
                        Binary  66   2     length of name in bytes
     ESD_RES_CLASS_PTR
                        Pointer 68   4     pointer to name string
3  ESD_ELEM_OFFSET      Binary  72   4   Offset within class element
                                           (LD, ER)
2  ESD_CLASS_OFFSET     Binary  76   4   Offset within class segment
                                           (ED, LD, PD, ER)
```

Notes:

1. This entry is ignored on input to the binder.
2. Recalculated by the binder.
3. Calculated on the ED and ER records, required input to LD.

Figure 35. Format for ESD entries (Part 3 of 3)

RLD entry (version 2)

The RLD Entry has been changed significantly from version 1.

```
    Field              Field  Off Leng        Description
    Name               Type   set

   IEWBRLD                                  Binder RLD buffer, Version 2
   RLDH_BUFFER_ID      Char     0   8       Buffer identifier "IEWBRLD"
   RLDH_BUFFER_LENG    Binary   8   4       Length of the buffer,
                                              including the header
   RLDH_VERSION        Binary  12   1       Version identifier
   *** RESERVED ***    Binary  13   3       Reserved, must be zeros
   RLDH_ENTRY_LENG     Binary  16   4       Length of each entry
   RLDH_ENTRY_COUNT    Binary  20   4       Number of entries in the buffer
   *** RESERVED ***    Binary  24   8       Reserved, must be zeros
   RLDH_ENTRY_ORIGIN           32           First RLD entry
   RLD_ENTRY                                RLD entry
   RLD_TYPE            Binary   0   1       Adcon type
       X'10'                                  Branch type (V-con)
       X'20'                                  Nonbranch type (A-con)
       X'21'                                  Address of class segment
       X'30'                                  Pseudoregister (Q-con)
       X'40'                                  Class or PRV length (CXD)
       X'60'                                  Relative immediate type
       X'80'                                  Long_Displacement 20-bit
                                              DL/DH offset (QY-con)
2  RLD_STATUS          Binary   1   1       Adcon status
       X'01'                                  References an unresolved symbol
       X'02'                                  References a resolved symbol
       X'03'                                  References a nonrelocatable
                                                symbol
1  RLD_RES_IDENT       Name     2   6       Name of section containing adcon
                                            ("P-pointer")
     RLD_RESIDENT_CHARS Binary  2   2         length of name in bytes
     RLD_RESIDENT_PTR  Pointer  4   4         pointer to name string
   RLD_ADCON_LENG      Binary   8   2       Adcon length
   RLD_RES_CLASS       Name    10   6       Name of class containing adcon
     RLD_CLASS_CHARS   Binary  10   2         length of name in bytes
     RLD_CLASS_PTR     Pointer 12   4         pointer to name string
   RLD_ELEM_OFFSET     Binary  16   4       Offset of the address constant
                                              within the containing element
2  RLD_CLASS_OFFSET    Binary  20   4       Offset of the address constant
                                              within the class segment
2  RLD_ADCON_BDY       Binary  24   1       Binder no longer maintains
                                            Adcon boundary status and
                                            changes this value to 0
   RLD_BIND_ATTR       Bit     25   1       Bind attributes
       1... ....                              Relocation sign (direction)
                                                '0'=positive, '1'=negative
       .1.. ....                              Set high order bit from
                                                AMODE of target
       ..1. ....                              Scope of reference
                                                '0'=internal, '1'=external
       ...1 ....                              High order bit of adcon
                                                reset by binder
```

Figure 36. Format for RLD entries (Part 1 of 2)

```
 Field              Field  Off Leng          Description
 Name               Type   set

RLD_XATTR_CLASS     Name    26   6   Extended attributes class
  RLD_XATTR_CLASS_CHARS
                    Binary  26   2     length of name in bytes
  RLD_XATTR_CLASS_PTR
                    Pointer 28   4     pointer to name string
RLD_XATTR_OFFSET    Binary  32   4   Extended attributes element offset
RLD_NAME_SPACE      Binary  36   1   Name space of reference
    X'00'                             Types 21, 40
    X'01'                             External reference (10, 20)
    X'02'                             Parts and pseudoregisters (30)
    X'03'-X'07'                       Available to language products
*** RESERVED ***    Binary  37   1   Reserved, must be zeros
RLD_TARGET          Name    38   6   Name of referenced symbol (for
                                        external references and Q-cons)
                                        or class (internal references
                                        and class references and lengths)
                                        ("R-Pointer")
  RLD_TARGET_CHARS  Binary  38   2     length of name in bytes
  RLD_TARGET_PTR    Pointer 40   4     pointer to name string
```

Notes:

1. Ignored on input.
2. Recalculated by the binder.

Figure 36. Format for RLD entries (Part 2 of 2)

Binder name list (version 2)

The name list has changed from version 1. Certain attributes, such as class length, bind control flags and element count, have been added to the name entries returned in the buffer.

```
    Field              Field  Off Leng          Description
    Name               Type   set

   IEWBBNL                                Binder Name List buffer, Ver 2
   BNLH_BUFFER_ID      Char     0   8     Buffer identifier "IEWBBNL"
   BNLH_BUFFER_LENG    Binary   8   4     Length of the buffer,
                                             including the header
   BNLH_VERSION        Binary  12   1     Version identifier
   *** RESERVED ***    Binary  13   3     Reserved, must be zeros
   BNLH_ENTRY_LENG     Binary  16   4     Length of each entry in the list
   BNLH_ENTRY_COUNT    Binary  20   4     Number of entries in the buffer
   *** RESERVED ***    Binary  24   8     Reserved, initialize to zeros
   BNLH_ENTRY_ORIGIN           32         First namelist entry

1  BNL_ENTRY                              Namelist Entry
   BNL_CLS_LENGTH      Binary   0   4     Class segment length

   BNL_BIND_FLAGS      Bit      4   1     Bind Attributes
      1... ....                            Generated by Binder
      .1.. ....                            No data present
      ..1. ....                            Varying length records
      ...1 ....                            Descriptive data (non-text)
      .... 1...                            Class has initial data
      .... .1..                            Fill character specified
      .... ...1                            Class validation error
2  BNL_PAD1            Bit      5  1    Loadability and alignment
                                         (NTYPE=CLASS only)
      0000 0000                            Not available
      ..1. ....                            A DEFER load class
      .10. ....                            A NOLOAD class
      .00. ....                            Class is initially loaded
      .... xxxx                            Boundary alignment as a power
                                           of 2, for example X'03' is
                                           2**3 = 8 byte align
   BNL_NAME            Name     6   6     Class or section name
     BNL_NAME_CHARS    Binary   6   2       length of name in bytes
     BNL_NAME_PTR      Pointer  8   4       pointer to name string
   BNL_ELEM_COUNT      Binary  12   4     Number of elements in class or
                                          section
```

Notes:

1. This entry is valid for output only.
2. For initial load classes, the offset of the class within a loaded module can be determined by padding the length of each segment as specified by the alignment, and accumulating the lengths in the order they are returned by GETN.

Figure 37. Format for binder name list entries

Module map (version 2)

The module map is new in version 2.

```
    Field              Field  Off Leng          Description
    Name               Type   set

   IEWBMAP                                  Module Map, Version 2
   MAPH_BUFFER_ID      Char     0   8   Buffer identifier "IEWBMAP"
   MAPH_BUFFER_LENG    Binary   8   4   Length of the buffer,
                                           including the header
   MAPH_VERSION        Binary  12   1   Version identifier
   *** RESERVED ***    Binary  13   3   Reserved, must be zeros
   MAPH_ENTRY_LENG     Binary  16   4   Length of each entry
   MAPH_ENTRY_COUNT    Binary  20   4   Number of entries in the list
   *** RESERVED ***    Binary  24   8   Reserved, initialize to zeros
   MAPH_ENTRY_ORIGIN           32       First map entry

1  MAP_ENTRY                             Map Entry
   MAP_NEXT            Binary   0   4   Offset of next sibling entry
   MAP_TYPE            Char     4   1   Type of map entry
       'M'                                Module
       'C'                                Class
       'O'                                Overlay segment
       'S'                                Section within class
       'P'                                Part within class
       'L'                                Label within section
       'X'                                Text extent within section
                                            or part
       'E'                                End-of-module
2  MAP_FLAGS           Bit      5   1   Flags
       1... ....                          Single extent implied (S)
       .1.. ....                          Loadable text class (C)
       ..1. ....                          Class is executable text (C)
       ...x xxxx                          Reserved
2  MAP_NAME            Name     6   6   Name of mapped element
                                          Null (M, X, E)
                                          region & segment (O)
     MAP_NAME_CHARS    Binary   6   2     length of name in bytes (O)
     MAP_NAME_PTR      Pointer  8   4     pointer to name string
         MAP_REGION    Binary   8   2       Region number (O)
         MAP_SEGMENT   Binary  10   2       Segment number (O)
2  MAP_OFFSET          Binary  12   4   Offset
                                          class offset (O, S, P)
                                          section offset (L, X)
                                          zero (M, C)
```

Figure 38. Format for module map list entries (Part 1 of 2)

```
    Field             Field  Off Leng          Description
    Name              Type   set

2  MAP_QUANTITY       Binary  16   4   Number of logical records/bytes
                                         Sum of all classes (M)
                                         Zero (L)
2  MAP_NAME_SPACE     Binary  20   1   Name space of label
       X'01'                             External Labels (L)
       X'02'                             Pseudoregisters (P)
       X'03'                             Parts (P)
       X'04'-X'07'                       Reserved
   *** RESERVED ***   Binary  21   1   Reserved.
2  MAP_RECORD_FMT     Binary  22   2   Class data format/record length
                                         Zero (M, E)
                                         Record length (C,O,S,L,P,X)
   *** RESERVED ***   Binary  24   8   Reserved.
```

Notes:
1. This entry is valid for output only.
2. Letters shown in parentheses refer to the map record type in which the flag is valid.

Figure 38. Format for module map list entries (Part 2 of 2)

Migration to version 2 buffers

Support for multiple text classes and a logical module structure in which sections are subdivided into classes requires some fundamental changes in the ESD. (See “Understanding binder programming concepts” on page 1 for a better understanding of module structure and content). At the highest level in the ESD hierarchy is the Section Definition (SD entry), which assigns some attributes to the section. At the next level is the Element Definition (ED entry), one for each class requested in the section. ED records are new and need not be present for binder-defined, nontext classes, such as ESD and IDR. Elements can contain either labels (LD entries) or parts (PR entries), depending upon the binding method for the class. Beginning with PM2, LD entries are required for all entry points: An SD entry is no longer an implied entry into offset zero of that section.

Two PM1 data classes have been extended in PM2 in order to properly reflect the multipart program object. Class B_ESD has been extensively modified, with some data moving from the SD record to lower level records, such as the ED and LD. In addition, new data fields have been added to support future expansion. Class B_RLD has also been modified for multipart modules, although not as extensively as the ESD. These changes must be considered when converting your program from version 1 buffers and API calls to version 2.

If your program was designed for version 1 buffers, you may not need to migrate to version 2 (or a later version) if you are processing data only from traditional (load module compatible) modules. However if a PM2 (or later level) program object exploits new function, it might contain data that cannot be converted to version 1 buffer format. In this case, you will receive an error message. In addition, the binder fast data API supports only PM1 format program objects with version 1 buffers.

If you want to convert your program to use the version 2 buffer formats and API calls, you must consider the following:

1. The SD ESD records no longer contain the length and offset of a section (since the section may now have more than one piece of text, in different text classes). The lengths and offsets are now in the ED ESD records.
2. The number of ESD record types has been reduced to six by combining all section-related records (for example, PC, CM, DS, ST, ET) into a single SD record type. The type qualifier can be used to distinguish between the various types of SD entries.
3. The ED record is new and is required for all text and compiler-defined classes, including ADATA.
4. An LD record is required for each entry point into the section. External references are not bound to SD entries in PM2.
5. PR and PD formats are identical. PRs, which represent a reference to a part or pseudoregister in a *merge* class, are created by the language translator and remain with the referencing section in the module. PD entries are always created during binding and are stored in a special summary section. The generated PD entries are the true definers of these parts, and contain the correct class offset, length and alignment.
6. Offsets relative to the start of the module or section in version 1 buffers have been replaced by class and element offsets, respectively, in Version 2.
7. Extended attributes information can be identified in certain ESD and RLD record types. Those data items are stored in the program object, but not processed in any way by the binder.
8. Name space has been added to both ESD and RLD records, since program symbols can be defined and referenced in multiple name spaces.
9. A new RLD type is defined for address constants that are to be set to the virtual address of a class.

The following tables show the correspondence between data elements in the two versions of the ESD and RLD buffers. Format codes shown have the following meaning:

Bnn Bit string, width in bits
Cnn Character data, width in bytes
Fnn Fixed point (integer) data, width in bits.
N06 Name, consisting of halfword name length followed by pointer to string. The IEWBUFF macro generates separate DS entries for the name length and pointer.
PTR 4-byte pointer

New data elements are normally set to zero when processing a PM1 module. This is not true if the new value is determined from other data present in the record.

Field correspondence for ESD records

Table 88. Comparison of new and old ESD formats. New and old ESD formats are compared, field-by-field. Numbers in the note column refer to notes in the conversion list which follows the table. An "=" indicates that the field name, format, contents and meaning are identical.

Version 2 Fields		Version 1 Fields		Ref
ESD_TYPE	C02	ESD_TYPE	C02	1
ESD_TYPE_QUAL	C02	ESD_TYPE	C02	2
ESD_NAME_SPACE	F08	(none)		3
ESD_SCOPE	C01	ESD_SCOPE	C01	=

Table 88. Comparison of new and old ESD formats (continued). New and old ESD formats are compared, field-by-field. Numbers in the note column refer to notes in the conversion list which follows the table. An "=" indicates that the field name, format, contents and meaning are identical.

Version 2 Fields		Version 1 Fields		Ref
ESD_NAME	N06	ESD_NAME	N06	=
ESD_RES_SECTION	N06	ESD_RESIDENT	N06	=
ESD_LENG	F31	ESD_LENG	F31	=
ESD_ALIGN	F08	ESD_ALIGN	F08	4
ESD_USABILITY	B08	ESD_USABILITY	B08	4
ESD_AMODE	B08	ESD_AMODE	B08	4
ESD_RMODE	B08	ESD_RMODE	B08	4
ESD_RECORD_FMT	F15	(none)		5
ESD_LOAD_FLAGS	B08	(none)		6
RO	B01	STORAGE	B08	7
NL	B01	(none)		8
MOVE	B01	(none)		9
SHR	B01	(none)		9
ESD_BIND_FLAGS	B08	(none)		10
GEND	B01	(none)		11
NO_CLASS	B01	(none)		12
VL	B01	(none)		13
DESCR	B01	(none)		14
ESD_BIND_CNTL	B08	(none)		15
METH	B02	(none)		15
TEXT_TYPE	B04	TEXT	C01	15
ESD_XATTR_CLASS	N06	(none)		16
ESD_XATTR_OFFSET	F31	(none)		16
ESD_SEGMENT	F15	ESD_SEGMENT	F15	=
ESD_REGION	F15	ESD_REGION	F15	=
ESD_SIGNATURE	C08	(none)		17
ESD_AUTOCALL	F08	ESD_AUTOCALL	F08	=
ESD_STATUS	B08	ESD_STATUS	B08	=
ESD_TGT_SECTION	N06	ESD_TARGET	N06	=
ESD_RES_CLASS	N06	(none)		18
ESD_ELEM_OFFSET	F31	ESD_SECTION_OFFSET	PTR	19
ESD_CLASS_OFFSET	F31	ESD_MODULE_OFFSET	PTR	19

ESD conversion notes (and PM1-PM2 differences)

The numbered items in the following list correspond to the numbers in the Ref column in the above table.

1. ESD_TYPE in PM2 supports only six record types: SD, ED, LD, PD, PR and ER. The PM1 ESD type values have been moved to the ESD_TYPE_QUAL field.
2. ESD_TYPE_QUAL contains the PM1 ESD type value, and is used to identify special behavioral characteristics. Routines that process all sections in the same way can now refer to ESD_TYPE only; those that must discriminate between the various section types must also look at the type qualifier.
3. ESD_NAME_SPACE is a new data element in PM2. It replaces some logic in bind processing and, as a result, gives more binding flexibility to the languages. For binder purposes, a name space is a set of symbols consisting of characters from the binder's character set and contains no duplicate definitions. In the case of a PM1 program object, the buffer is set to one of the following values:
   - 0 - SD and ED entries.
   - 1 - LD and ER entries, and PD/PR entries for external data items.
   - 2 - PD/PR entries representing pseudoregisters and other parts
4. Alignment, Usability, RMODE and AMODE are unchanged from PM1 values, except that some have been moved to lower level ESD structures:
   - Alignment to ED. PR alignment comes from old PR record.
   - AMODE to LD. Each external label can have its own AMODE, but can be overridden by a referencing ALIAS specification.
   - RMODE to ED. This is necessary for multipart loading.

   All remain as valid SD fields for compatibility, although they are copied to the other record types prior to binding.
5. ESD_RECORD_FMT is a new data element in PM2. It is used only on the ED record type, and indicates the structure and record length of the data contained in the class being described by the ED record. As such it becomes a class attribute, against which the record length field in any PUTD buffers for that class is validated.
6. ESD_LOAD_FLAGS is a new data element in PM2. Its purpose is to enable multiclass loading by the program management Loader, and together with RMODE contains the various attributes the loader needs for proper placement of the class segment in storage. Load flags are extracted from input of all sources.
7. The PM1 format defines ESD_STORAGE value X'10' as representing read-only storage. This field is reduced to a single load flag in PM2, RO, where B'1' means read-only.
8. Loader attribute NL, if set, indicates that the class segment is not to be loaded with the module.
9. These are new PM2 data elements.
10. Bind flags ESD_BIND_FLAGS are new PM2 data elements. In a PM2 program object, they represent binding attributes that are derived from other sources during binding. They control the binding and output processes of the binder. Four flags have been defined, GEND, NO_CLASS, VL and DESCR.
11. GEND indicates that the defined entity has been generated by the binder. It is used with:
    - SD records for SEGTABs and ENTABs and special sections (X'00000001' and X'00000003').
    - ED records for B_IDRB and B_MAP classes.
    - LD records created by the binder for each CSECT.
    - PD summary records created by the binder for parts.
12. NO_CLASS indicates that there is no initial data in this class. The element is filled with the fill byte, if specified, during bind and/or load.

13. VL indicates that the byte stream data consists of varying length records, each preceded by a halfword length field.
14. DESCR indicates whether the class contains text or descriptive data. Text is byte stream data that can contain or be the target of address constants. Descriptive data is everything else. Both text and descriptive data could be either loaded or not.
15. ESD_BIND_CNTL contains two data elements needed to control the binding process. Unlike ESD_BIND_FLAGS, this data is provided by the language translator. This field consists of two sub-elements, METH and TEXT_TYPE.
    - Binder control field METH identifies the binding method to be used on the class. It is specified only on the ED record, and should be set to B'00' on other record types for PM2 program objects.
    - The field ESD_TEXT has been reduced to a half-byte field under the name TEXT_TYPE. The data is meaningful only for external references, on the ER and LD records, and allows cursory interface matching during binding. The values represent the type of text that is defined by the label (LD) or expected by the reference (ER) and are:
      – 0 - Unspecified. If either the LD and/or ER have this value, the interface is accepted by the binder.
      – 1 - Data. For data references.
      – 2 - Instructions. For calls.
      – 3-15 - Undefined value for special translator usage.
16. These are new PM2 data elements. ESD_XATTR_CLASS and ESD_XATTR_OFFSET locate extended attributes for a label (LD) or external reference (ER). On SD, ED, PR and PD records, these fields are meaningless and are set to zero.

    IBM language products are not currently using signatures, extended attributes, or interface matching.
17. SIGNATURE is a new data element in PM2. It allows quick interface matching between reference and definition during the binding process, avoiding the invocation of a time-consuming interface validation exit if the signatures match or are not specified.

    IBM language products are not currently using signatures, extended attributes or interface matching.
18. This is a new PM2 data element. For a PM2 program object, CLASS is displayed on all LD, PD and ER records and should be set to zero on SD and ED. For LD, PD and ER records, it is the data class containing the label or part.
19. Change of field names: “section” becomes “element”, “module” becomes “class”.

Table 89. ESD element usage. Table shows which elements are displayed, by ESD record type. For usage of sub-elements, refer to individual record type formats.

Element Name	SD	ED	LD	PR	ER
ESD_NAME	R	R	R	R	R
ESD_RES_SECT	O	O	O	O	O
ESD_TARGET					C
ESD_RES_CLASS			R	R	C
ESD_XATTR_CLASS			O		O
ESD_XATTR_OFFSET			O		O
ESD_LENG		R		R	

Table 89. ESD element usage (continued). Table shows which elements are displayed, by ESD record type. For usage of sub-elements, refer to individual record type formats.

Element Name	SD	ED	LD	PR	ER
ESD_ELEM_OFFSET			R		C
ESD_CLASS_OFFSET		C	C	C	C
ESD_REGION	C				
ESD_SEGMENT	C				
ESD_TYPE	R	R	R	R	R
ESD_TYPE_QUAL	R				O
ESD_SCOPE			R		R
ESD_NAME_SPACE		R	C	C	R
ESD_RECORD_FMT		R			
ESD_LOAD_FLAGS		R			
ESD_BIND_FLAGS	C	R	C		
ESD_BIND_CNTL		R	R		R
ESD_ALIGN	O	R		R	
ESD_USABILITY	R				
ESD_RMODE		R			
ESD_AMODE	O		R		
ESD_AUTOCALL					C
ESD_STATUS					C
ESD_SIGNATURE					C
Note: R = Required on input, displayed on output; O = Optional on input, displayed on output; C = Recalculated by Bind.					

Field correspondence for RLD records

Table 90. Comparison of new and old RLD formats. New and old workmod RLD formats are compared, field-by-field. Numbers in the notes column refer to notes in the conversion list that follows the table. An "=" in this column indicates that the two formats are identical in name, format, content and meaning.

Version 2 Fields		Version 1 Fields		Ref
RLD_TYPE	F08	RLD_TYPE	F08	1
RLD_STATUS	F08	RLD_STATUS	F08	=
RLD_RES_IDENT	N06	RLD_RESIDENT	N06	=
RLD_ADCON_LENG	F15	RLD_ADCON_LENG	F15	=
RLD_RES_CLASS	N06	(none)		2
RLD_ELEM_OFFSET	F31	RLD_SECTION_OFFSET	F31	3
RLD_CLASS_OFFSET	F31	RLD_MODULE_ OFFSET	F31	3
RLD_ADCON_BDY	F08	RLD_ADCON_BDY	F08	=
RLD_BIND_FLAGS	B08	(none)		4
DIRECTION	B01	RLD_ADCON_ DIRECTION	F08	5
HOBSET	B01	(none)		4

Table 90. Comparison of new and old RLD formats (continued). New and old workmod RLD formats are compared, field-by-field. Numbers in the notes column refer to notes in the conversion list that follows the table. An "=" in this column indicates that the two formats are identical in name, format, content and meaning.

Version 2 Fields		Version 1 Fields		Ref
INT_EXT	B01	(none)		4
HOBCHG	B01	(none)		4
RLD_XATTR_CLASS	N06	(none)		4
RLD_XATTR_OFFSET	F31	(none)		4
RLD_NAME_SPACE	F08	(none)		6
RLD_TARGET	N06	RLD_TARGET	N06	=

RLD conversion notes (and PM1-PM2 differences)

The numbered items in the following list correspond to the numbers in the Ref column in the above table.

1. This is equivalent to the PM1 field RLD_TYPE. A new RLD type added in PM2, type 21, which is used to address the origin of any loaded class segment.
2. Name of the class containing the adcon. For PM1 program objects, RLD_CLASS is set to B_TEXT.
3. Module and section offset have been replaced with the offset relative to the start of the class and element, respectively.
4. New PM2 field. Is initialized to zero unless otherwise noted.
5. The 8-bit PM1 direction flag is reduced to a single bind flag in PM2.
6. For A- and V-type adcons (types 10 & 20), RLD_NAME_SPACE=1. For Q-type adcons (type 30), RLD_NAME_SPACE=2. Otherwise, RLD_NAME_SPACE=0.

Version 3 buffer formats

VERSION=3 must be specified on the IEWBUFF macro to obtain buffers in this format.

Version 3 buffers support new program management features introduced in PM3. Most buffers have not changed except for the version number, but version 3 is supported by IEWBUFF for all classes, whether or not there is a format change. Differences between versions 2 and 3 include:

- ESD buffer has changed slightly to support DLLs and C/C++ semantic s.
- RLD buffer has changed slightly to support RLDs resident in parts.
- A new binder-defined class, B_PARTINIT, has a new buffer format named IEWBPTI.

This section contains the buffer layouts that have changed or have been added in version 3. As indicated in Table 85 on page 240, all the other buffer formats will be the same for version 3 as in version 2.

ESD entry (version 3)

Please note that the version 3 ESD buffer has been slightly changed from the version 2 buffer.

```
 Field              Field  Off Leng          Description
 Name               Type   set

IEWBESD                                  Binder ESD buffer, Version 3
ESDH_BUFFER_ID      Char     0   8   Buffer identifier "IEWBESD"
ESDH_BUFFER_LENG    Binary   8   4   Length of the buffer,
                                        including the header
ESDH_VERSION        Binary  12   1   Version identifier
*** RESERVED ***    Binary  13   3   Reserved, must be zeros
ESDH_ENTRY_LENG     Binary  16   4   Length of each entry
ESDH_ENTRY_COUNT    Binary  20   4   Number of entries in the buffer
*** RESERVED ***    Binary  24   8   Reserved
ESDH_ENTRY_ORIGIN           32       First ESD entry

ESD_ENTRY                                ESD entry
ESD_TYPE            Char     0   2   ESD Type
    C'  '                                Null Entry
    C'SD'                                Control Section
    C'LD'                                Label Definition
    C'ER'                                External Reference
    C'PR'                                Part Reference
    C'PD'                                Part Definition
    C'ED'                                Element Definition

ESD_TYPE_QUAL       Char     2   2   ESD Type Qualifier
    C' '                                 (no qualification)
    C'SD'                                Section Definition (SD)
    C'CM'                                Common (SD)
    C'ST'                                Segment Table (SD)
    C'ET'                                Entry Table (SD)
    C'PC'                                Unnamed Section (SD)
    C'PR'                                Part Reference (PR, PD)
    C'PD'                                Part Definition (PR, PD)
    C'ER'                                External Reference (ER)
    C'WX'                                Weak Reference (ER)
ESD_NAME_SPACE      Binary   4   1   Name Space for symbols
    X'00'                                  Class and section names (SD, ED)
    X'01'                                  Labels and references (LD, ER)
    X'02'                                  Pseudoregisters (PR, PD)
    X'03'                                  Parts (PR, PD) in merge classes
    X'04'-x'07'                            Reserved
ESD_SCOPE           Char     5   1   Scope of Name
    ' '                                    Not applicable
    'S'                                    Section (Types SD/private,ST,ET)
    'M'                                    Module (Types SD/CSECT,LD,
                                                  ER/weak,CM,PR,DS,PD)
    'L'                                    Library (Type ER/strong)
    'X'                                    Symbol can be IMPORTED or
                                                  EXPORTED.
ESD_NAME            Name     6   6   Symbol represented by ESD record
  ESD_NAME_CHARS    Binary   6   2     length of name in bytes
  ESD_NAME_PTR      Pointer  8   4     pointer to name string
ESD_SYMBOL_ATTR     Binary  12   1   Symbol attributes
    1... ....                              ON  = strong ref or def.
                                           OFF = weak ref or def.
    .1.. ....                              ON = this symbol can be renamed
    ..1. ....                              ON = Symbol is a descriptor
    ...1 ....                              ON = symbol is a C++ mangled name
    .... 1...                              ON = symbol uses XPLINK linkage
                                                conventions
    .... .1..                              Environment exists
    .... ..11                              ** Reserved **
```

Figure 39. Format for ESD entries (Part 1 of 3)

```
    Field              Field  Off Leng          Description
    Name               Type   set

 ESD_FILL_CHAR       Char    13   1   Value to fill with
1  ESD_RES_SECTION     Name    14   6   Name of containing section
     ESD_RESIDENT_CHARS Binary 14   2     length of name in bytes
     ESD_RESIDENT_PTR  Pointer 16   4     pointer to name string
   ESD_LENG            Binary  20   4   Length of defined element
                                        (ED, PD, PR)
ESD_ALIGN           Binary  24   1   Alignment specification from
                                        language processor.  Indicates
                                        Alignment of section contribution
                                        within class segment (ED, PD, PR)
       X'00'                              Byte alignment (PD, PR)
       X'01'                              Halfword (PD, PR)
       X'02'                              Fullword (PD, PR)
       X'03'                              Doubleword (PD, PR, ED)
       X'04'                              Quadword (PR, PD, ED)
       X'0C'                              4K page (ED)
   ESD_USABILITY       Binary  25   1   Reusability of Section (SD)
       X'00'                              Unspecified
       X'01'                              Nonreusable
       X'02'                              Reusable
       X'03'                              Reentrant
       X'04'                              Refreshable
   ESD_AMODE           Bit     26   1   Addressing Mode for Section or
                                        label (SD, LD)
       X'00'                              Unspecified
       X'01'                              AMODE 24
       X'02'                              AMODE 31
       X'03'                              AMODE ANY (24 or 31)
       X'04'                              AMODE MIN
       X'05'                              Unused, reserved
       X'06'                              AMODE 64
   ESD_RMODE           Bit     27   1   Residence Mode for class element
                                          (ED))
       X'01'                              RMODE 24
       X'03'                              RMODE ANY (24 or 31)
   ESD_RECORD_FMT      Binary  28   2   Record format for class (ED)
       H'1'                               Byte stream
       H'>1'                              Fixed length records
    Field              Field  Off Leng          Description
    Name               Type   set

   ESD_LOAD_FLAGS      Bit     30   1   Load Attributes (ED)
       1... ....                          Read-only
       .1.. ....                          Do not load with module
       ..1. ....                          Moveable
       ...1 ....                          Shareable
       .... 1...                          Deferred
       .... .111                          Reserved
   ESD_BIND_FLAGS      Bit     31   1   Bind Attributes
2      1... ....                          Binder generated (SD, ED, LD)
2      .1.. ....                          No class data available (ED)
2      ..1. ....                          Variable length records (ED)
       ...1 ....                          Descriptive data (not text) (ED)
       .... 1...                          ON = class contains part initializers (ED)
       .... .1..                          ON = fill character has been specified
                                                    (ED)
       .... ..1.                          Class has padding
       .... ...1                          ** Reserved **
```

Figure 39. Format for ESD entries (Part 2 of 3)

```
    Field              Field  Off Leng          Description
    Name               Type   set

   ESD_BIND_CNTL       Bit     32   1   Bind control information
1      1... ....                         Removable class(ED)
1      .x.. ....                         ** Reserved **
1      ..xx ....                         Binding method (ED)
1      ..00 ....                           CAT (Catenated text)
1      ..01 ....                           MRG (Merged parts)
1      ..1x ....                           ** Reserved **
   *** RESERVED ***    Char    33   1   Reserved
   ESD_XATTR_CLASS     Name    34   6   Extended attributes class
                                        (LD, ER)
   ESD_XATTR_CLASS_CHARS
                       Binary  34   2     length of name in bytes
   ESD_XATTR_CLASS_PTR
                       Pointer 36   4     pointer to name string
   ESD_XATTR_OFFSET    Binary  40   4   Extended attributes element offset
                                        (LD, ER)
2  ESD_SEGMENT         Binary  44   2   Overlay segment number (SD)
2  ESD_REGION          Binary  46   2   Overlay region number (SD)
   ESD_SIGNATURE       Char    48   8   Interface signature
2  ESD_AUTOCALL        Binary  56   1   Autocall specification (ER)
       1... ....                          ** Reserved **
       .1.. ....                          Entry in LPA.  If ON, name is
                                          an alias.
       ..xx xxxx                          ** Reserved **
2  ESD_STATUS          Bit     57   1   Resolution status (ER)
       1... ....                          Symbol is resolved
       .1.. ....                          Processed by autocall
       ..1. ....                          INCLUDE attempted
       ...1 ....                          Member not found
       .... 1...                          Resolved outside module
       .... .1..                          NOCALL or NEVERCALL
       .... ..1.                          No strong references
       .... ...1                          Special call library
2  ESD_TGT_SECTION     Name    58   6   Target section (ER)
     ESD_TARGET_CHARS  Binary  58   2     length of name in bytes
     ESD_TARGET_PTR    Pointer 60   4     pointer to name string
   *** RESERVED ***    Char    64   2   Reserved
   ESD_RES_CLASS       Name    66   6   Name of containing class
                                        (LD, PD) or target class (ER)
     ESD_RES_CLASS_CHARS
                       Binary  66   2     length of name in bytes
     ESD_RES_CLASS_PTR
                       Pointer 68   4     pointer to name string
3  ESD_ELEM_OFFSET     Binary  72   4   Offset within class element
                                        (LD, ER)
2  ESD_CLASS_OFFSET    Binary  76   4   Offset within class segment
                                        (ED, LD, PD, ER)
   *** RESERVED ***    Char    80   2   Reserved
   ESD_ADA_CHARS       Binary  82   2   Environment name length (LD)
   ESD_ADA_PTR         Pointer 84   4   Pointer to name of environment (LD)
   *** RESERVED ***    Char    88   4   Reserved
   ESD_PRIORITY        Binary  92   4   Binding priority
```

Notes:

1. This entry is ignored on input to the binder.
2. Recalculated by the binder.
3. Calculated on the ED and ER records, required input to LD.

Figure 39. Format for ESD entries (Part 3 of 3)

RLD entry (version 3)

The RLD Entry has been changed slightly from version 2.

```
    Field              Field  Off Leng        Description
    Name               Type   set

   IEWBRLD                                  Binder RLD buffer, Version 3
   RLDH_BUFFER_ID      Char     0   8   Buffer identifier "IEWBRLD"
   RLDH_BUFFER_LENG    Binary   8   4   Length of the buffer,
                                           including the header
   RLDH_VERSION        Binary  12   1   Version identifier
   *** RESERVED ***    Binary  13   3   Reserved, must be zeros
   RLDH_ENTRY_LENG     Binary  16   4   Length of each entry
   RLDH_ENTRY_COUNT    Binary  20   4   Number of entries in the buffer
   *** RESERVED ***    Binary  24   8   Reserved, must be zeros
   RLDH_ENTRY_ORIGIN           32       First RLD entry
   RLD_ENTRY                            RLD entry
   RLD_TYPE            Binary   0   1   Adcon type
       X'10'                              Branch type (V-con)
       X'20'                              Nonbranch type (A-con)
       X'21'                              Address of class segment
       X'30'                              Pseudoregister (Q-con)
       X'40'                              Class or PRV length (CXD)
1      X'50'                            Loader token
       X'60'                            Relative immediate type
       X'70'                            Reference to environment
       X'80'                              Long_Displacement 20-bit
                                            DL/DH offset (QY-con)
2  RLD_STATUS          Binary   1   1   Adcon status
       X'01'                              References an unresolved symbol
       X'02'                              References a resolved symbol
       X'03'                              References a nonrelocatable
                                            symbol
1  RLD_RES_IDENT       Name     2   6   Name of section containing adcon
                                        ("P-pointer")
     RLD_RESIDENT_CHARS Binary  2   2     length of name in bytes
     RLD_RESIDENT_PTR  Pointer  4   4     pointer to name string
   RLD_ADCON_LENG      Binary   8   2   Adcon length
   RLD_RES_CLASS       Name    10   6   Name of class containing adcon
     RLD_CLASS_CHARS   Binary  10   2     length of name in bytes
     RLD_CLASS_PTR     Pointer 12   4     pointer to name string
   RLD_ELEM_OFFSET     Binary  16   4   Offset of the address constant
                                           within the containing element
2  RLD_CLASS_OFFSET    Binary  20   4   Offset of the address constant
                                           within the class segment
2  *** RESERVED ***    Binary  24   1   Reserved

   RLD_BIND_ATTR      Bit     25   1   Bind attributes
       1... ....                          Relocation sign (direction)
                                            '0'=positive, '1'=negative
       .1.. ....                          Set high order bit from
                                            AMODE of target
       ..1. ....                          Scope of reference
                                            '0'=internal, '1'=external
       ...1 ....                          High order bit of adcon
                                            reset by binder
       .... 1...                          Adcon is resident in a part (PD)
       .... ..1.                          Target of adcon marked XPLINK
|      .... ...1                          Adcon is part of a group of conditional
|                                          sequential adcons
```

Figure 40. Format for RLD entries (Part 1 of 2)

Field Name	Field Type	Off set	Leng	Description
RLD_XATTR_CLASS	Name	26	6	Extended attributes class
RLD_XATTR_CLASS_CHARS	Binary	26	2	length of name in bytes
RLD_XATTR_CLASS_PTR	Pointer	28	4	pointer to name string
RLD_XATTR_OFFSET	Binary	32	4	Extended attributes element offset
RLD_NAME_SPACE	Binary	36	1	Name space of reference
X'00'				Types 21, 40
X'01'				External reference (10, 20)
X'02'				Pseudoregisters (30)
X'03'				Parts (PR,PD) in writeable static
X'04'-X'07'				Available to language products
*** RESERVED ***	Binary	37	1	Reserved, must be zeros
RLD_TARGET	Name	38	6	Name of referenced symbol (for external references and Q-cons) or class (internal references and class references and lengths) ("R-Pointer")
RLD_TARGET_CHARS	Binary	38	2	length of name in bytes
RLD_TARGET_PTR	Pointer	40	4	pointer to name string
*** RESERVED ***	Binary	44	2	Reserved, must be zeros
RLD_RES_PART	Name	46	6	Name of resident part.
RLD_RES_PART_CHARS	Binary	46	2	length of name in bytes
RLD_RES_PART_PTR	Pointer	48	4	pointer to name string

Notes:
1. Ignored on input.
2. Recalculated by the binder.

Figure 40. Format for RLD entries (Part 2 of 2)

PARTINIT entry (version 3)

The PARTINIT entry has been added for version 3.

Field Name	Field Type	Off set	Leng	Description
IEWBPTI				Binder PARTINIT buffer, Version 3
PTIH_BUFFER_ID	Char	0	8	Buffer identifier "IEWBPTI"
PTIH_BUFFER_LENG	Binary	8	4	Length of the buffer, including the header
PTIH_VERSION	Binary	12	1	Version identifier
*** RESERVED ***	Binary	13	3	Reserved, must be zeros
PTIH_ENTRY_LENG	Binary	16	4	Length of each entry
PTIH_ENTRY_COUNT	Binary	20	4	Number of entries in the buffer
*** RESERVED ***	Binary	24	8	Reserved, must be zeros
PTIH_ENTRY_ORIGIN		32		First PARTINIT entry
PTI_ENTRY				PARTINIT entry
PTI_DATA_LENG	Binary	0	2	Record length
PTI_DATA_REP	Binary	2	2	Repetition factor
*** RESERVED ***	Binary	4	2	Reserved, must be zeros
PTI_DATA_CLASS_LL	Binary	6	2	Class name length
PTI_DATA_CLASS_PTR	Pointer	8	4	Pointer to class name
PTI_DATA_NAME_LL	Binary	12	2	Part Initializer name length
PTI_DATA_NAME_PTR	Pointer	14	4	Pointer to part initializer name
PTI_DATA_OFFSET	Pointer	18	4	Placement of data within part
PTI_DATA_BYTE	Undef.	22	var	Beginning of data (the length varies from 1 to a maximum which takes into account the combined size of the buffer header, the buffer names list area, and the entry header; while the actual size for each entry will depend on its content, the maximum given a single entry (PTIH_ENTRY_COUNT = 1) is BYTES-1078 or SIZE depending which keyword is used)

Figure 41. Format for PARTINIT entries

Migration to version 3 buffers

Version 3 buffers are very similar to Version 2 buffers. The change from Version 2 to Version 3 is much smaller than the change from Version 1 to Version 2. Therefore, if you are currently using Version 1 buffers, see the discussion in “Migration to version 2 buffers” on page 261 before reading this section.

If your program was designed for PM2 format, it should continue to work without change. Fast data will convert a PM3 program object to PM2 (but not to PM1).

Part initializers

Part Initializer buffers are not supported in version 1 or 2. Part initializers contain text data to be placed in parts (defined by PR or PD ESD records) in merge classes.

ESD conversion notes

The ESD records contain the following new fields:

1. ESD_SYMBOL_ATTR
   a. *strong or weak*

      If definition - a strong definition will override a weak definition. Multiple weak definitions are allowed.

If reference - unresolved weak reference will not result in an error message.

b. *renameable* symbol is eligible for renaming under control of UPCASE option, RENAME control statements, and C run-time library renaming algorithms.

c. *descriptor* symbol represents a linkage descriptor.

2. *ESD_LOAD_FLAGS* is a new attribute flag (DEFER) to indicate a deferred load class has been defined. This bit should be interpreted in conjunction with the do-not-load flag (NL) as follows:

NL	DEFER	MEANING
0	0	INITIAL LOAD
0	1	DEFERRED LOAD
1	0	NO LOAD

3. The *ESD_PRIORITY* field can be used to order the PRs in a merge class. The lowest priority parts in the output module will be assigned the smallest offset.

RLD conversion notes

RLD_RES_PART can be set to specify the name of the resident part for an RLD that describes an adcon within a part. If RLD_RES_PART is set, RLD_ELEM_OFFSET must be interpreted as the offset of the adcon from the beginning of the part. The part name must match the name on a PR or PD type ESD record.

Version 4 buffer formats

Version 4 buffers are those supported in OS/390 DFSMS Version 2 Release 10 program management (PM3). VERSION=4 must be specified on the IEWBUFF macro to obtain buffers in this format. Most buffers have not changed except for the version number, but VERSION=4 is supported by IEWBUFF for all buffers, whether or not there is a format change.

The name list has changed from version 2. Binder Name List buffer has has been extended to provide additional information about segments within a class. Segment ID and segment offset fields have been added. Boundary alignment, which was added late to the version 2/3 format as a temporary expedient, has been removed because segment offset provides better information.

This section contains the buffer layouts that have changed or have been added in Version 4. As indicated in Table 85 on page 240, all the other buffer formats will be the same for Version 4 as in Version 3.

Binder name list (version 4)

Please note that the version 4 binder name list buffer has been slightly changed from the version 3 buffer.

```
    Field              Field  Off Leng          Description
    Name               Type   set

   IEWBBNL                                Binder Name List buffer, Version 4
   BNLH_BUFFER_ID      Char     0   8     Buffer identifier "IEWBBNL"
   BNLH_BUFFER_LENG    Binary   8   4     Length of the buffer,
                                             including the header
   BNLH_VERSION        Binary  12   1     Version identifier
   *** RESERVED ***    Binary  13   3     Reserved, must be zeros
   BNLH_ENTRY_LENG     Binary  16   4     Length of each entry in the list
   BNLH_ENTRY_COUNT    Binary  20   4     Number of entries in the buffer
   *** RESERVED ***    Binary  24   8     Reserved, initialize to zeros
   BNLH_ENTRY_ORIGIN           32         First namelist entry

1  BNL_ENTRY                              Namelist Entry
   BNL_CLS_LENGTH      Binary   0   4     Class segment length (NTYPE=CLASS only)

   BNL_BIND_FLAGS      Bit      4   1     Bind Attributes (NTYPE=CLASS only)
      1... ....                            Generated by Binder
      .1.. ....                            No data present
      ..1. ....                            Varying length records
      ...1 ....                            Descriptive data (non-text)
      .... 1...                            Class has initial data
      .... .1..                            Fill character specified
      .... ...1                            Class validation error
   BNL_LOAD_FLAGS      Bit      5   1     Loadability (NTYPE=CLASS only)
      1... ....                            Read Only
      .01. ....                            A DEFER load class
      .10. ....                            A NOLOAD class
      .00. ....                            Class is initially loaded
      ...1 ....                            Sharable
      .... 1...                            Moveable (AdCon free)
   BNL_NAME_CHARS      Binary   6   2     Length of name
   BNL_NAME_PTR        Pointer  8   4     Pointer to class or section
                                          name
   BNL_ELEM_COUNT      Binary  12   4     Number of elements in class
                                          or section
   BNL_SEGM_ID         Binary  16   2     Segment ID (NTYPE=CLASS only)
   BNL_PAD1            Binary  18   2     Reserved, set to zero
   BNL_SEGM_OFF        Binary  20   4     Class offset from start
                                          of segment (NTYPE=CLASS only)
```

Notes:

1. This entry is valid for output only.

Figure 42. Format for binder name list entries

Version 5 buffer formats

Version 5 buffers are first supported in z/OS Version 1 Release 3 program management (PM4). VERSION=5 must be specified on the IEWBUFF macro to obtain buffers in this format. Most buffers have not changed, but VERSION=5 is supported by IEWBUFF for all buffers, whether or not there is a format change.

The PMAR is a buffer format added in version 5. It is the primary conduit of information from the binder to the program loader. The PMAR is mapped by the system mapping macro IEWPMAR. The PMAR buffer format provided by IEWBUFF only reserves the appropriate amount of storage without defining its layout. For additional information about IEWPMAR, see *z/OS MVS Data Areas* in z/OS Internet Library at http://www.ibm.com/systems/z/os/zos/bkserv/.

One additional flag byte, ESD_ATTRIBUTES, has been defined in the ESD buffer, using a previously reserved field. The version 5 layout is defined in Figure 44 on page 278.

PMAR entry (version 5)

Field Name	Field Type	Off set	Leng	Description
IEWBPMR				Binder PMAR buffer, Version 5
PMRH_BUFFER_ID	Char	0	8	Buffer identifier "IEWBPMR"
PMRH_BUFFER_LENG	Binary	8	4	Length of the buffer, including the header
PMRH_VERSION	Binary	12	1	Version identifier
*** RESERVED ***	Binary	13	3	Reserved, must be zeros
PMRH_ENTRY_LENG	Binary	16	4	Length of entries (always 1)
PMRH_ENTRY_COUNT	Binary	20	4	Number of entries (bytes) in the buffer
*** RESERVED ***	Binary	24	8	Reserved, initialize to zeros
PMRH_ARRAY	Undef.	32	var	PMAR data (length varies from 1 to 1024 bytes depending on value in PMRH_ENTRY_COUNT; typically, PM4 has 136 bytes, PM3 has 128 bytes, PM2 has 112 bytes, and PM1 has 88 bytes of PMAR

Figure 43. Format for PMAR entries

ESD entry (version 5)

```
 Field              Field  Off Leng          Description
 Name               Type   set

IEWBESD                                  Binder ESD buffer, Version 5
ESDH_BUFFER_ID      Char     0   8   Buffer identifier "IEWBESD"
ESDH_BUFFER_LENG    Binary   8   4   Length of the buffer,
                                        including the header
ESDH_VERSION        Binary  12   1   Version identifier (Constant 2)
*** RESERVED ***    Binary  13   3   Reserved, must be zeros
ESDH_ENTRY_LENG     Binary  16   4   Length of each entry
ESDH_ENTRY_COUNT    Binary  20   4   Number of entries in the buffer
*** RESERVED ***    Binary  24   8   Reserved
ESDH_ENTRY_ORIGIN           32       First ESD entry

ESD_ENTRY                                ESD entry
ESD_TYPE            Char     0   2   ESD Type
    C'  '                                Null Entry
    C'SD'                                Control Section
    C'LD'                                Label Definition
    C'ER'                                External Reference
    C'PR'                                Part Reference
    C'PD'                                Part Definition
    C'ED'                                Element Definition

ESD_TYPE_QUAL       Char     2   2   ESD Type Qualifier
    C' '                                 (no qualification)
    C'SD'                                Section Definition (SD)
    C'CM'                                Common (SD)
    C'ST'                                Segment Table (SD)
    C'ET'                                Entry Table (SD)
    C'PC'                                Unnamed Section (SD)
    C'PR'                                Part Reference (PR, PD)
    C'PD'                                Part Definition (PR, PD)
    C'ER'                                External Reference (ER)
    C'WX'                                Weak Reference (ER)
ESD_NAME_SPACE      Binary   4   1   Name Space for symbols
    X'00'                                  Class and section names (SD, ED)
    X'01'                                  Labels and references (LD, ER)
    X'02'                                  Pseudoregisters (PR, PD)
    X'03'                                  Parts(PR, PD) in merge classes
    X'04'-x'07'                            Reserved
ESD_SCOPE           Char     5   1   Scope of Name
    ' '                                    Not applicable
    'S'                                    Section (Types SD/private,ST,ET)
    'M'                                    Module (Types SD/CSECT,LD,
                                                  ER/weak,CM,PR,DS,PD)
    'L'                                    Library (Type ER/strong)
    'X'                                    Symbol can be IMPORTED or
                                                  EXPORTED.
ESD_NAME            Name     6   6   Symbol represented by ESD record
  ESD_NAME_CHARS    Binary   6   2     length of name in bytes
  ESD_NAME_PTR      Pointer  8   4     pointer to name string
```

Figure 44. Format for ESD entries (Part 1 of 4)

```
    Field              Field  Off Leng          Description
    Name               Type   set

   ESD_SYMBOL_ATTR     Binary  12   1   Symbol attributes
       1... ....                           ON  = strong ref or def.
                                           OFF = weak ref or def.
       .1.. ....                           ON = this symbol can be renamed
       ..1. ....                           ON = Symbol is a descriptor
       ...1 ....                           ON = symbol is a C++ mangled name
       .... 1...                           ON = symbol uses XPLINK linkage
                                                   conventions
       .... .1..                           Environment exists
       .... ..11                           ** Reserved **
   ESD_FILL_CHAR       Char    13   1   Value to fill with
1  ESD_RES_SECTION     Name    14   6   Name of containing section
     ESD_RESIDENT_CHARS Binary 14   2     length of name in bytes
     ESD_RESIDENT_PTR  Pointer 16   4     pointer to name string
   ESD_LENG            Binary  20   4   Length of defined element
                                          (ED, PD, PR)
   ESD_ALIGN           Binary  24   1   Alignment specification from
                                          language processor.  Indicates
                                          Alignment of section contribution
                                          within class segment (ED, PD, PR)
       X'00'                              Byte alignment (PD, PR)
       X'01'                              Halfword (PD, PR)
       X'02'                              Fullword (PD, PR)
       X'03'                              Doubleword (PD, PR, ED)
      X'04'                              Quadword (PR, PD, ED)
       X'0C'                              4K page (ED)
   ESD_USABILITY       Binary  25   1   Reusability of Section (SD)
       X'00'                              Unspecified
       X'01'                              Nonreusable
       X'02'                              Reusable
       X'03'                              Reentrant
       X'04'                              Refreshable
```

Figure 44. Format for ESD entries (Part 2 of 4)

```
     Field              Field  Off Leng          Description
     Name               Type   set

   ESD_AMODE            Bit     26   1   Addressing Mode for Section or
                                           label (SD, LD)
       X'00'                                 Unspecified
       X'01'                                 AMODE 24
       X'02'                                 AMODE 31
       X'03'                                 AMODE ANY (24 or 31)
       X'04'                                 AMODE MIN
       X'05'                                 Unused, reserved
       X'06'                                 AMODE 64
   ESD_RMODE            Bit     27   1   Residence Mode for class element
                                             (ED)
       X'01'                                 RMODE 24
       X'03'                                 RMODE ANY (24 or 31)
       X'04'                                 RMODE 64   ESD_RECORD_FMT
                        Binary  28   2   Record format for class (ED)
       H'1'                                  Byte stream
       H'>1'                                 Fixed length records
   ESD_LOAD_FLAGS       Bit     30   1   Load Attributes (ED)
       1... ....                             Read-only
       .1.. ....                             Do not load with module
       ..1. ....                             Moveable
       ...1 ....                             Shareable
       .... 1...                             Deferred
       .... .111                             Reserved
   ESD_BIND_FLAGS       Bit     31   1   Bind Attributes
2      1... ....                             Binder generated (SD, ED, LD)
2      .1.. ....                             No class data available (ED)
2      ..1. ....                             Variable length records (ED)
       ...1 ....                             Descriptive data (not text) (ED)
       .... 1...                             ON = class contains part initializers (ED)
       .... .1..                             ON = fill character has been specified (ED)
       .... ..1.                             Class has padding
       .... ...1                             ** Reserved **
   ESD_BIND_CNTL        Bit     32   1   Bind control information
1      1... ....                             Removable class(ED)
1      .x.. ....                             ** Reserved **
1      ..xx ....                             Binding method (ED)
1      ..00 ....                               CAT (Catenated text)
1      ..01 ....                               MRG (Merged parts)
1      ..1x ....                               ** Reserved **
     ESD_ATTRIBUTES     Bit     33   1   General attributes
   4   1... ....                              Compiled as system LE
   4   .1.. ....                              Compiled as lightweight LE
       ..11 ....                              ** Reserved **
   4   .... xx..                              Error severity for dups (PD, LD)
       .... 00..                              - I-level
       .... 01..                              - W-level
       .... 10..                              - E-level
       .... 11..                              - S-level
       .... ..1.                              ** Reserved **
       .... ...1                              ** Reserved **
```

Figure 44. Format for ESD entries (Part 3 of 4)

```
    Field              Field  Off Leng          Description
    Name               Type   set

   ESD_XATTR_CLASS     Name    34   6   Extended attributes class
                                        (LD, ER)
     ESD_XATTR_CLASS_CHARS
                       Binary  34   2     length of name in bytes
     ESD_XATTR_CLASS_PTR
                       Pointer 36   4     pointer to name string
   ESD_XATTR_OFFSET    Binary  40   4   Extended attributes element offset
                                        (LD, ER)
2  ESD_SEGMENT         Binary  44   2   Overlay segment number (SD)
2  ESD_REGION          Binary  46   2   Overlay region number (SD)
   ESD_SIGNATURE       Char    48   8   Interface signature
2  ESD_AUTOCALL        Binary  56   1   Autocall specification (ER)
       1... ....                          ** Reserved **
       .1.. ....                          Entry in LPA.  If ON, name is
                                          an alias.
       ..xx xxxx                          ** Reserved **
2  ESD_STATUS          Bit     57   1   Resolution status (ER)
       1... ....                          Symbol is resolved
       .1.. ....                          Processed by autocall
       ..1. ....                          INCLUDE attempted
       ...1 ....                          Member not found
       .... 1...                          Resolved outside module
       .... .1..                          NOCALL or NEVERCALL
       .... ..1.                          No strong references
       .... ...1                          Special call library
2  ESD_TGT_SECTION     Name    58   6   Target section (ER)
     ESD_TARGET_CHARS  Binary  58   2     length of name in bytes
     ESD_TARGET_PTR    Pointer 60   4     pointer to name string
   *** RESERVED ***    Char    64   2   Reserved
   ESD_RES_CLASS       Name    66   6   Name of containing class
                                        (LD, PD) or target class (ER)
     ESD_RES_CLASS_CHARS
                       Binary  66   2     length of name in bytes
     ESD_RES_CLASS_PTR
                       Pointer 68   4     pointer to name string
3  ESD_ELEM_OFFSET     Binary  72   4   Offset within class element
                                        (LD, ER)
2  ESD_CLASS_OFFSET    Binary  76   4   Offset within class segment
                                        (ED, LD, PD, ER)
   *** RESERVED ***    Char    80   2   Reserved
   ESD_ADA_CHARS       Binary  82   2   Environment name length (LD)
   ESD_ADA_PTR         Pointer 84   4   Pointer to name of environment (LD)
   *** RESERVED ***    Char    88   4   Reserved
   ESD_PRIORITY        Binary  92   4   Binding priority
```

Notes:
1. This entry is ignored on input to the binder.
2. Recalculated by the binder.
3. Calculated on the ED and ER records, required input to LD.
4. Valid for SDs only.

Figure 44. Format for ESD entries (Part 4 of 4)

Version 6 buffer formats

Version 6 buffers are supported in z/OS Version 1 Release 5. CUI and LIB are buffer formats added in version 6. The CUI buffer is used to get compile unit information and LIB is used to return library path information. In addition, the BNL buffer has changed.

LIB entry (version 6)

Field Name	Field Type	Off set	Leng	Description
IEWBLIB				Binder LIB buffer, Version 6
LIBH_BUFFER_ID	Char	0	8	Buffer identifier "IEWBLIB"
LIBH_BUFFER_LENG	Binary	8	4	Length of the buffer, including the header
LIBH_VERSION	Binary	12	1	Version identifier
*** RESERVED ***	Binary	13	3	Reserved, must be zeros
LIBH_ENTRY_LENG	Binary	16	4	Length of entries (always 1)
LIBH_ENTRY_COUNT	Binary	20	4	Number of entries (bytes) in the buffer
*** RESERVED ***	Binary	24	8	Reserved, initialize to zeros
LIB_ARRAY	Undef.	32	var	Library Path data (length varies from 1 to 1024 bytes, depending on value in LIBH_ENTRY_COUNT)

Figure 45. Format for LIB entries

CUI entry (version 6)

Field Name	Field Type	Off set	Leng	Description
IEWBCUI				Binder CUI buffer, Version 6
CUIH_BUFFER_ID	Char	0	8	Buffer identifier "IEWBCUI"
CUIH_BUFFER_LENG	Binary	8	4	Length of the buffer, including the header
CUIH_VERSION	Binary	12	1	Version identifier
*** RESERVED ***	Binary	13	3	Reserved, must be zeros
CUIH_ENTRY_LENG	Binary	16	4	Length of entries
CUIH_ENTRY_COUNT	Binary	20	4	Number of entries (bytes) in the buffer
*** RESERVED ***	Binary	24	8	Reserved, initialize to zeros
CUIH_ENTRY_ORIGIN		32		First compile unit entry
CUI_ENTRY				Compile Unit Entry
CUI_CU	Binary	0	4	Compile unit number
CUI_SOURCE_CU	Binary	4	4	Source of compile unit number
*** RESERVED ***	Binary	8	2	Reserved, must be zero
CUI_MEMBER_LEN	Binary	10	2	Length of member
CUI_MEMBER_PTR	Pointer	12	4	Pointer to the member
*** RESERVED ***	Binary	16	2	Reserved, must be zero
CUI_PATH_LEN	Binary	18	2	Length of path
CUI_PATH_PTR	Pointer	20	4	Pointer to the path
*** RESERVED ***	Binary	24	2	Reserved, must be zero
CUI_DSNAME_LEN	Binary	26	2	Length of dsname
CUI_DSNAME_PTR	Pointer	28	4	Pointer to the dsname
CUI_DDNAME	Char	32	8	Ddname
*** RESERVED ***	Binary	40	2	Reserved, must be zero
CUI_CONCAT	Binary	42	1	Concat
CUI_TYPE	Binary	43	1	Source type
X'00'				Load module
X'01'				Generated by PUTD API version 1
X'02'				Generated by PUTD API version 2 or higher
X'10'				PO1 (PM1) format program object
X'11'				Object module (traditional format)
X'12'				Object module (XOBJ format)
X'13'				Object module (GOFF format)
X'14'				Unknown
X'15'				Workmod
X'1E'				Generated by the binder
X'20'				PO2 (PM2) format program object
X'30'				PO3 (PM3) format program object
X'41'				PO4 (PM4) format program object, z/OS 1.3 compatible
X'42'				z/OS 1.5 compatible
X'43'				z/OS 1.7 compatible
X'51'				PO5 (PM5) format program object, z/OS 1.8 compatible
*** RESERVED ***	Binary	44	4	Reserved, must be zero
*** RESERVED ***	Binary	48	2	Reserved, must be zero
CUI_C_MEMBER_LEN	Binary	50	2	Length of member (original)
CUI_C_MEMBER_PTR	Pointer	52	4	Pointer to the member (original)
*** RESERVED ***	Binary	56	2	Reserved, must be zero
CUI_C_PATH_LEN	Binary	58	2	Length of path (original)
CUI_C_PATH_PTR	Pointer	60	4	Pointer to the path (original)
*** RESERVED ***	Binary	64	2	Reserved, must be zero
CUI_C_DSNAME_LEN	Binary	66	2	Length of dsname (original)
CUI_C_DSNAME_PTR	Pointer	68	4	Pointer to the dsname
*** RESERVED ***	Char	72	3	Reserved, must be zero
CUI_C_TYPE	Binary	75	1	Source type within the object file
CUI_C_SEQ	Binary	76	4	CU sequence number within the object file

Figure 46. Format for CUI entries

Notes:

1. The header record contains information about the target workmod as a whole. The format is the same as that of the other records. CUI_CU, CUI_SOURCE_CU and CUI_C_SEQ will always be zero in the header record.
2. Fields that have the comment original refer to the object module used to build the program object the first time. These field contents are not changed if the program object is rebound. However, this information is available only if the

program object is compatible with the z/OS 1.5 format or higher version. Thus, complete information is returned for nonheader records only if the CUI_TYPE value is 42 or greater. CUI_TYPE for program objects in formats compatible with releases earlier than z/OS 1.5. Load modules have CUI_TYPE set to the format of the target module.

3. Entries for compile units representing sections created by the binder, including section 1, contain no information other than the compile unit number (CUI_CU) and the type (CUI_TYPE).

Binder name list (version 6)

Please note that the version 6 binder name list buffer has been slightly changed from the version 4 buffer.

```
    Field              Field  Off Leng          Description
    Name               Type   set

   IEWBBNL                                Binder Name List buffer, Version 6
   BNLH_BUFFER_ID      Char     0   8     Buffer identifier "IEWBBNL"
   BNLH_BUFFER_LENG    Binary   8   4     Length of the buffer,
                                             including the header
   BNLH_VERSION        Binary  12   1     Version identifier
   *** RESERVED ***    Binary  13   3     Reserved, must be zeros
   BNLH_ENTRY_LENG     Binary  16   4     Length of each entry in the list
   BNLH_ENTRY_COUNT    Binary  20   4     Number of entries in the buffer
   *** RESERVED ***    Binary  24   8     Reserved, initialize to zeros
   BNLH_ENTRY_ORIGIN           32         First namelist entry

1  BNL_ENTRY                              Namelist Entry
2  BNL_CLS_LENGTH      Binary   0   4     Class segment length (NTYPE=CLASS only)
3  BNL_SECT_CU         Binary   0   4     Compile unit number (NTYPE=SECTION only)
   BNL_BIND_FLAGS      Bit      4   1     Bind Attributes (NTYPE=CLASS only)
      1... ....                            Generated by Binder
      .1.. ....                            No data present
      ..1. ....                            Varying length records
      ...1 ....                            Descriptive data (non-text)
      .... 1...                            Class has initial data
      .... .1..                            Fill character specified
      .... ...1                            Class validation error
   BNL_LOAD_FLAGS      Bit      5   1     Loadability (NTYPE=CLASS only)
      1... ....                            Read Only
      .01. ....                            A DEFER load class
      .10. ....                            A NOLOAD class
      .00. ....                            Class is initially loaded
      ...1 ....                            Sharable
      .... 1...                            Moveable (AdCon free)
   BNL_NAME_CHARS      Binary   6   2     Length of name
   BNL_NAME_PTR        Pointer  8   4     Pointer to class or section
                                          name
   BNL_ELEM_COUNT      Binary  12   4     Number of elements in class
                                          or section
   BNL_SEGM_ID         Binary  16   2     Segment ID (NTYPE=CLASS only)
   BNL_PAD1            Binary  18   2     Reserved, set to zero
   BNL_SEGM_OFF        Binary  20   4     Class offset from start
                                          of segment (NTYPE=CLASS only)
```

Notes:

1. This entry is valid for output only.
2. BNL_SECT_CU is at the same offset as BNL_CLS_LENGTH.

Figure 47. Format for binder name list entries

Version 7 buffer formats

Version 7 buffers are supported in z/OS Version 1 Release 10.

Language processor identification data (version 7)

	Field Name	Field Type	Off set	Leng	Description
	IEWBIDL				Language Processor IDR Record, Version 7
	IDLH_BUFFER_ID	Char	0	8	Buffer identifier "IEWBIDL"
	IDLH_BUFFER_LENG	Binary	8	4	Length of the buffer, including the header
	IDLH_VERSION	Binary	12	1	Version identifier
	*** RESERVED ***	Binary	13	3	Reserved, must be zeros
	IDLH_ENTRY_LENG	Binary	16	4	Length of each entry
	IDLH_ENTRY_COUNT	Binary	20	4	Number of entries in the buffer
	*** RESERVED ***	Binary	24	8	Reserved, must be zeros
	IDLH_ENTRY_ORIGIN		32		First IDR entry
4	IDL_ENTRY				Language IDR Entry
	IDL_PID_ID	Char	0	10	Translator product ID
	IDL_VERSION	Char	10	2	Version of the translator
	IDL_MOD_LEVEL	Char	12	2	Modification level of translator
2	IDL_DATE_PROCESSED	Char	14	7	Date compiled or assembled (yyyyddd)
3	IDL_TIME_PROCESSED	Char	21	9	Time compiled or assembled (hhmmssttt)
1	IDL_RESIDENT	Name	30	6	The name of the section to which this IDR data applies.
	IDL_RESIDENT_CHARS	Binary	30	2	length of name in bytes
	IDL_RESIDENT_PTR	Pointer	32	4	pointer to name string

Notes:

1. Ignored on input to the binder.
2. If the module is saved as a load module, dates are truncated to 5 characters (YYDDD).
3. If the module is saved as a load module, the time stamp is discarded.
4. There is only one IDRL entry for each section. However, if the section has been produced by a multistage compilation, there is one IDRL record for each processor involved in the generation of the object code.

Figure 48. Format for IDRL entries

Binder name list (version 7)

Please note that the version 7 binder name list buffer has been slightly changed from the version 6 buffer.

```
     Field              Field  Off Leng          Description
     Name               Type   set

    IEWBBNL                                Binder Name List buffer, Version 7
    BNLH_BUFFER_ID      Char     0   8     Buffer identifier "IEWBBNL"
    BNLH_BUFFER_LENG    Binary   8   4     Length of the buffer,
                                             including the header
    BNLH_VERSION        Binary  12   1     Version identifier
    *** RESERVED ***    Binary  13   3     Reserved, must be zeros
    BNLH_ENTRY_LENG     Binary  16   4     Length of each entry in the list
    BNLH_ENTRY_COUNT    Binary  20   4     Number of entries in the buffer
    *** RESERVED ***    Binary  24   8     Reserved, initialize to zeros
    BNLH_ENTRY_ORIGIN           32         First namelist entry

1   BNL_ENTRY                              Namelist Entry
2   BNL_CLS_LENGTH      Binary   0   4     Class segment length (NTYPE=CLASS only)
3   BNL_SECT_CU         Binary   0   4     Compile unit number (NTYPE=SECTION only)
    BNL_BIND_FLAGS      Bit      4   1     Bind Attributes(NTYPE=CLASS only)
       1... ....                            Generated by Binder
       .1.. ....                            No data present
       ..1. ....                            Varying length records
       ...1 ....                            Descriptive data (non-text)
       .... 1...                            Class has initial data
       .... .1..                            Fill character specified
       .... ...1                            Class validation error
    BNL_LOAD_FLAGS      Bit      5   1     Loadability (NTYPE=CLASS only)
       1... ....                            Read Only
       .xx. ....                            Time of load
       .00. ....                            Class is initially loaded
       .01. ....                            A DEFER load class
       .10. ....                            A NOLOAD class
       ...1 ....                            Sharable
       .... 1...                            Moveable (AdCon free)
       .... ..xx                            Binding Method
       .... ..00                            CAT(catenated text)
       .... ..01                            MERGE(merged parts)
    BNL_NAME_CHARS      Binary   6   2     Length of name
    BNL_NAME_PTR        Pointer  8   4     Pointer to class or section
                                           name
    BNL_ELEM_COUNT      Binary  12   4     Number of elements in class
                                           or section
    BNL_SEGM_ID         Binary  16   2     Segment ID (NTYPE = CLASS only)
    BNL_ATTR            Bit     18   1
    BNL_ALIGN           Bit          5      Alignment (NTYPE = CLASS only)
      00011                                 Doubleword
      00100                                 Quadword
      01100                                 Page (4K)
    BNL_RMODE           Bit          3      Residence mode (NTYPE = CLASS only)
      001                                   Rmode 24
      011                                   Rmode ANY
      100                                   Rmode 64
    BNL_NAMESPACE       Binary  19   1     Namespace (NTYPE = CLASS only)
      x'01' = catenate class
      x'02' = pseudoregisters (merge class)
      x'03' = parts (merge class)
    BNL_SEGM_OFF        Binary  20   4     Class offset from start of segment
                                             (NTYPE=CLASS only)
```

Notes:

1. This entry is valid for output only.
2. BNL_SECT_CU is at the same offset as BNL_CLS_LENGTH.

Figure 49. Format for binder name list entries

Appendix E. Data areas

This topic contains general-use programming interface and associated guidance information.

This topic describes the formats of the following data areas.

- The first describes the “PDS directory entry format on entry to STOW.”
- The second describes the “PDS directory entry format returned by BLDL” on page 296.
- The third describes the “PDSE directory entry returned by DESERV (SMDE data area)” on page 303.
- The fourth describes “Accessing program object class information” on page 306.
- The last describes “Module Map” on page 308

PDS directory entry format on entry to STOW

The following describes the format of a partitioned data set (PDS) directory entry. It is returned in simulated 256-byte blocks when BSAM or QSAM is used to read a PDSE directory entry This format shows the control block on entry to the STOW macro. You can use the IHAPDS macro to map this control block.

Decimal Offsets	Hex Offsets	Type/Value	Len	Name (Dim)	Description
0	(0)	CHARACTER	8	PDS2NAME	- LOAD MODULE MEMBER NAME OR ALIAS
8	(8)	CHARACTER	3	PDS2TTRP	- TTR OF FIRST BLOCK OF NAMED MEMBER
11	(B)	BITSTRING	1	PDS2INDC	- INDICATOR BYTE
		1...		PDS2ALIS	“BIT0”- NAME IN THE FIRST FIELD IS AN ALIAS
		.11.		PDS2NTTR	“BIT1+ BIT2”- NUMBER OF TTR'S IN THE USER DATA FIELD
		...1 1111		PDS2LUSR	“BIT3+ BIT4+ BIT5+ BIT6+ BIT7” - LENGTH OF USER DATA FIELD IN HALF WORDS
12	(C)	CHARACTER	1	PDS2USRD (0)	- START OF VARIABLE LENGTH USER DATA FIELD
12	(C)	CHARACTER	3	PDS2TTRT	- TTR OF FIRST BLOCK OF TEXT
15	(F)	CHARACTER	1	PDS2ZERO	- ZERO
16	(10)	CHARACTER	3	PDS2TTRN	- TTR OF NOTE LIST OR SCATTER/TRANSLATION TABLE. USED FOR MODULES IN SCATTER LOAD FORMAT OR OVERLAY STRUCTURE ONLY.
19	(13)	SIGNED	1	PDS2NL	- NUMBER OF ENTRIES IN NOTE LIST FOR MODULES IN OVERLAY STRUCTURE
20	(14)	BITSTRING	2	PDS2ATR (0)	- TWO-BYTE MODULE ATTRIBUTE FIELD

© Copyright IBM Corp. 1991, 2012

Data areas

Decimal Offsets	Hex Offsets	Type/Value	Len	Name (Dim)	Description
20	(14)	BITSTRING	1	PDS2ATR1	- FIRST BYTE OF MODULE ATTRIBUTE FIELD
		1...		PDS2RENT	“BIT0”- REENTERABLE
		.1..		PDS2REUS	“BIT1”- REUSABLE
		..1.		PDS2OVLY	“BIT2”- IN OVERLAY STRUCTURE
		...1		PDS2TEST	“BIT3”- MODULE TO BE TESTED - TESTRAN
		.... 1...		PDS2LOAD	“BIT4”- ONLY LOADABLE
		.... .1..		PDS2SCTR	“BIT5”- SCATTER FORMAT
		.... ..1.		PDS2EXEC	“BIT6”- EXECUTABLE
		1		PDS21BLK	“BIT7”- IF ZERO, MODULE CONTAINS MULTIPLE RECORDS WITH AT LEAST ONE BLOCK OF TEXT. — IF ONE, MODULE CONTAINS NO RLD ITEMS AND ONLY ONE BLOCK OF TEXT.
21	(15)	BITSTRING	1	PDS2ATR2	- SECOND BYTE OF MODULE ATTRIBUTE FIELD
		1...		PDS2FLVL	“BIT0”- IF ZERO, MODULE CAN BE PROCESSED BY ALL LEVELS OF LINKAGE EDITOR. — IF ONE, MODULE CAN BE PROCESSED ONLY BY F LEVEL OF LINKAGE EDITOR.
		.1..		PDS2ORG0	“BIT1”- LINKAGE EDITOR ASSIGNED ORIGIN OF FIRST BLOCK OF TEXT IS ZERO.
		..1.		PDS2EP0	“BIT2”- ENTRY POINT ASSIGNED BY LINKAGE EDITOR IS ZERO
		...1		PDS2NRLD	“BIT3”- MODULE CONTAINS NO RLD ITEMS
		.... 1...		PDS2NREP	“BIT4”- MODULE CANNOT BE REPROCESSED BY LINKAGE EDITOR
		.... .1..		PDS2TSTN	“BIT5”- MODULE CONTAINS TESTRAN SYMBOL CARDS
		.... ..1.		PDS2LEF	“BIT6”- MODULE CREATED BY LINKAGE EDITOR F
		1		PDS2REFR	“BIT7”- REFRESHABLE MODULE
22	(16)	SIGNED	3	PDS2STOR	- TOTAL CONTIGUOUS VIRTUAL STORAGE REQUIREMENT OF MODULE
25	(19)	SIGNED	2	PDS2FTBL	- LENGTH OF FIRST BLOCK OF TEXT
27	(1B)	ADDRESS	3	PDS2EPA	- ENTRY POINT ADDRESS ASSOCIATED WITH MEMBER NAME OR WITH ALIAS NAME IF ALIAS INDICATOR IS ONE

Decimal Offsets	Hex Offsets	Type/Value	Len	Name (Dim)	Description
30	(1E)	ADDRESS	3	(0)	- LINKAGE EDITOR ASSIGNED ORIGIN OF FIRST BLOCK OF TEXT (when bit 0 is off)
30	(1E)	BITSTRING	3	PDS2FTBO (0)	- FLAG BYTES (VS/1-VS/2 USE OF FIELD)
30	(1E)	BITSTRING	1	PDS2FTB1	- BYTE 1 OF PDS2FTBO
		1...		PDSAOSLE	“BIT0”- MODULE HAS BEEN PROCESSED BY VS/1-VS/2 LINKAGE EDITOR
		.1..		PDS2BIG	“BIT1” THIS MODULE REQUIRES 16M OR MORE OF VIRTUAL STORAGE.
		..1.		PDS2PAGA	“BIT2”- PAGE ALIGNMENT REQUIRED FOR LOAD MODULE
		...1		PDS2SSI	“BIT3”- SSI INFORMATION PRESENT
		.... 1...		PDSAPFLG	“BIT4”- INFORMATION IN PDSAPF IS VALID
		.... .1..		PDS2LFMT	“BIT5”- MODULE IS IN PROGRAM OBJECT FORMAT. THE PDS2FTB3 FIELD IS VALID AND CONTAINS ADDITIONAL FLAGS
31	(1F)	BITSTRING	1	PDS2FTB2	- BYTE 2 OF PDS2FTBO
		1...		PDS2ALTP	“BIT0:”- ALTERNATE PRIMARY FLAG. INDICATES THE PRIMARY NAME WAS GENERATED BY THE BINDER.
		...1		PDSLRMOD	“BIT3:”- PROGRAM RESIDENCE MODE (1=RMODE=ANY, 0=RMODE=24)
		.... 11..		PDSAAMOD	“BIT4+ BIT5”- ALIAS ENTRY POINT ADDRESSING MODE (00=AMODE=24, 10=AMODE=31, 01=AMODE=64, 11=AMODE=ANY)
		11		PDSMAMOD	“BIT6+ BIT7”- MAIN ENTRY POINT ADDRESSING MODE (SAME BIT SETTINGS AS PDSAAMOD)
32	(20)	BITSTRING	1	PDS2RLDS (0)	NUMBER OF RLD/CONTROL RECORDS WHICH FOLLOW THE FIRST BLOCK OF TEXT
32	(20)	BITSTRING	1	PDS2FTB3	- BYTE 3 OF PDS2FTBO
		1...		PDS2NMIG	“BIT0”- THIS PROGRAM OBJECT LOAD MODULE CANNOT BE CONVERTED TO RECORD FORMAT
		.1..		PDS2PRIM	“BIT1”- FETCHOPT PRIME WAS SPECIFIED
		..1.		PDS2PACK	“BIT2”- FETCHOPT PACK WAS SPECIFIED
		X'21'		PDSBCEND	“*”- END OF BASIC SECTION
		X'21'		PDSBCLN	“PDSBCEND-PDS2”- LENGTH OF BASIC SECTION

THE FOLLOWING SECTION IS FOR LOAD MODULES WITH SCATTER LOAD

		X'21'		PDSS01	"*"- START OF SCATTER LOAD SECTION
33	(21)	SIGNED	2	PDS2SLSZ	- NUMBER OF BYTES IN SCATTER LIST
35	(23)	SIGNED	2	PDS2TTSZ	- NUMBER OF BYTES IN TRANSLATION TABLE
37	(25)	CHARACTER	2	PDS2ESDT	- IDENTIFICATION OF ESD ITEM (ESDID) OF CONTROL SECTION TO WHICH FIRST BLOCK OF TEXT BELONGS
39	(27)	CHARACTER	2	PDS2ESDC	- IDENTIFICATION OF ESD ITEM (ESDID) OF CONTROL SECTION CONTAINING ENTRY POINT
		X'29'		PDSS01ND	"*"- END OF SCATTER LOAD SECTION
		X'8'		PDSS01LN	"PDSS01ND-PDSS01"- LENGTH OF SCATTER LOAD SECTION

THE FOLLOWING SECTION IS FOR LOAD MODULES WITH ALIAS NAMES

		X'29'		PDSS02	"*"- START OF ALIAS SECTION
41	(29)	ADDRESS	3	PDS2EPM	- ENTRY POINT FOR MEMBER NAME
		X'29'		DEENTBK	"PDS2EPM"— ALIAS
44	(2C)	CHARACTER	8	PDS2MNM	- MEMBER NAME OF LOAD MODULE. WHEN THE FIRST FIELD (PDS2NAME) IS AN ALIAS NAME, THIS FIELD CONTAINS THE ORIGINAL NAME OF THE MEMBER EVEN AFTER THE MEMBER HAS BEEN RENAMED.
		X'34'		PDSS02ND	"*"- END OF ALIAS SECTION
		X'B'		PDSS02LN	"PDSS02ND-PDSS02"- LENGTH OF ALIAS SECTION

THE FOLLOWING SECTION IS FOR SSI INFORMATION AND IS ON A HALFWORD BOUNDARY

52	(34)	SIGNED	2	PDSS03 (0)	- FORCE HALFWORD ALIGNMENT FOR SSI SECTION
52	(34)	CHARACTER	4	PDSSSIWD (0)	- SSI INFORMATION WORD
52	(34)	SIGNED	1	PDSCHLVL	- CHANGE LEVEL OF MEMBER
53	(35)	BITSTRING	1	PDSSSIFB	- SSI FLAG BYTE
		.1..		PDSFORCE	"BIT1"- A FORCE CONTROL CARD WAS USED WHEN EXECUTING THE IHGUAP PROGRAM
		..1.		PDSUSRCH	"BIT2"- A CHANGE WAS MADE TO MEMBER BY THE INSTALLATION, AS OPPOSED TO AN IBM-DISTRIBUTED CHANGE

		...1		PDSEMFIX	“BIT3”- SET WHEN AN EMERGENCY IBM-AUTHORIZED PROGRAM 'FIX' IS MADE, AS OPPOSED TO CHANGES THAT ARE INCLUDED IN AN IBM-DISTRIBUTED MAINTENANCE PACKAGE
		.... 1...		PDSDEPCH	“BIT4”- A CHANGE MADE TO THE MEMBER IS DEPENDENT UPON A CHANGE MADE TO SOME OTHER MEMBER IN THE SYSTEM
		.... .11.		PDSSYSGN	“BIT5+ BIT6”- FLAGS THAT INDICATE WHETHER OR NOT A CHANGE TO THE MEMBER WILL NECESSITATE A PARTIAL OR COMPLETE REGENERATION OF THE SYSTEM
		....		PDSNOSGN	“X'00'”- NOT CRITICAL FOR SYSTEM GENERATION
		.... ..1.		PDSCMSGN	“BIT6”- MAY REQUIRE COMPLETE REGENERATION
		.... .1..		PDSPTSGN	“BIT5”- MAY REQUIRE PARTIAL REGENERATION
		1		PDSIBMMB	“BIT7”- MEMBER IS SUPPLIED BY IBM
54	(36)	CHARACTER	2	PDSMBRSN	- MEMBER SERIAL NUMBER
		X'38'		PDSS03ND	“*”- END OF SSI SECTION
		X'4'		PDSS03LN	“PDSS03ND-PDSS03”- LENGTH OF SSI SECTION

THE FOLLOWING SECTION IS FOR APF INFORMATION

		X'38'		PDSS04	“*”- START OF APF SECTION
56	(38)	CHARACTER	2	PDSAPF (0)	- PROGRAM AUTHORIZATION FACILITY (APF) FIELD
56	(38)	SIGNED	1	PDSAPFCT	- LENGTH OF PROGRAM AUTHORIZATION CODE (PDSAPFAC) IN BYTES
57	(39)	CHARACTER	1	PDSAPFAC	- PROGRAM AUTHORIZATION CODE
		X'3A'		PDSS04ND	“*”- END OF APF SECTION
		X'2'		PDSS04LN	“PDSS04ND-PDSS04”- LENGTH OF APF SECTION

THE FOLLOWING SECTION IS FOR LARGE PROGRAM OBJECTS (LPO)

		X'3A'		PDSLPO	“*”- START OF LPO SECTION LENGTH
58	(3A)	SIGNED	1	PDS2LPOL	- LPO SECTION LENGTH
59	(3B)	SIGNED	4	PDS2VSTR	- VIRTUAL STORAGE REQUIREMENT FOR THIS MODULE
63	(3F)	SIGNED	4	PDS2MEPA	- MAIN ENTRY POINT OFFSET

67	(43)	SIGNED	4	PDS2AEPA	- ALIAS ENTRY POINT OFFSET. ONLY VALID IF THIS IS AN DIRECTORY ENTRY IS FOR AN ALIAS
		X'47'		PDSLPOND	"*"- END OF LPO SECTION
		X'D'		PDSLPOLN	"PDSLPOND-PDSLPO"- LENGTH OF LPO SECTION

Cross reference

Name	Hex Offset	Hex Value
PDSAAMOD	1F	C
PDSAOSLE	1E	80
PDSAPF	38	
PDSAPFAC	39	
PDSAPFCT	38	
PDSAPFLG	1E	8
PDSBCEND	20	21
PDSBCLN	20	21
PDSCHLVL	34	
PDSCMSGN	35	2
PDSDEPCH	35	8
PDSEMFIX	35	10
PDSFORCE	35	40
PDSIBMMB	35	1
PDSLLM (alternate for PDSLPO)	3A	ƀ
PDSLLMLN (alternate for PDSLPOLN	47	D
PDSLLMND (alternate for PDSLPOND)	47	ƀ
PDSLPO	3A	ƀ
PDSLPOL	3A	D
PDSLPOLN	47	D
PDSLPOND	47	ƀ
PDSLRMOD	1F	10
PDSMAMOD	1F	3
PDSMBRSN	36	
PDSNOSGN	35	0
PDSPTSGN	35	4
PDSSSIFB	35	
PDSSSIWD	34	
PDSSYSGN	35	6
PDSS01	20	21
PDSS01LN	27	8
PDSS01ND	27	29
PDSS02	27	29
PDSS02LN	2C	B
PDSS02ND	2C	34
PDSS03	34	
PDSS03LN	36	4
PDSS03ND	36	38
PDSS04	36	38
PDSS04LN	39	2
PDSS04ND	39	3A
PDSUSRCH	5	20
PDS2AEPA	43	

Name	Hex Offset	Hex Value
PDS2ALIS	B	80
PDS2ALTP	1F	80
PDS2ATR	14	
PDS2ATR1	14	
PDS2ATR2	15	
PDS2BIG	1E	40
PDS2EPA	1B	
PDS2EPM	29	
PDS2EP0	15	20
PDS2ESDC	27	
PDS2ESDT	25	
PDS2EXEC	14	2
PDS2FLVL	15	80
PDS2FTBL	19	
PDS2FTBO	1E	
PDS2FTB1	1E	
PDS2FTB2	1F	
PDS2FTB3	20	
PDS2INDC	B	
PDS2LEF	15	2
PDS2LFMT	1E	4
PDS2LLML	3A	
PDS2LOAD	14	8
PDS2LUSR	B	1F
PDS2MEPA	3F	
PDS2MNM	2C	
PDS2NAME	0	
PDS2NL	13	
PDS2NMIG	20	80
PDS2NREP	15	8
PDS2NRLD	15	10
PDS2NTTR	B	60
PDS2ORG0	15	40
PDS2OVLY	14	20
PDS2PACK	20	20
PDS2PAGA	1E	20
PDS2PRIM	20	40
PDS2REFR	15	1
PDS2RENT	14	80
PDS2REUS	14	40
PDS2RLDS	20	
PDS2SCTR	14	4
PDS2SLSZ	21	
PDS2SSI	1E	10
PDS2STOR	16	
PDS2TEST	14	10
PDS2TSTN	15	4
PDS2TTRN	10	
PDS2TTRP	8	
PDS2TTRT	C	
PDS2TTSZ	23	
PDS2USRD	C	
PDS2VSTR	3B	

Data areas

Name	Hex Offset	Hex Value
PDS2ZERO	F	
PDS21BLK	14	1

PDS directory entry format returned by BLDL

The following describes the format of a partitioned data set (PDS) directory entry. This format describes what is returned by BLDL for each PDS directory entry member. You can use the IHAPDS macro to map this control block.

Decimal Offsets	Hex Offsets	Type/Value	Len	Name (Dim)	Description
0	(0)	CHARACTER	8	PDS2NAME	- LOAD MODULE MEMBER NAME OR ALIAS
8	(8)	CHARACTER	3	PDS2TTRP	- TTR OF FIRST BLOCK OF NAMED MEMBER
11	(B)	CHARACTER	1	PDS2CNCT	- CONCATENATION NUMBER OF THE DATA SET
12	(C)	CHARACTER	1	PDS2LIBF	- LIBRARY FLAG FIELD
		....		PDS2LNRM	“X'00'”- NORMAL CASE
		1		PDS2LLNK	“X'01'”- IF DCB OPERAND IN BLDL MACRO INSTRUCTION WAS SPECIFIED AS ZERO, NAME WAS FOUND IN LINK LIBRARY
		.... ..1.		PDS2LJOB	“X'02'”- IF DCB OPERAND IN BLDL MACRO INSTRUCTION WAS SPECIFIED AS ZERO, NAME WAS FOUND IN JOB LIBRARY
13	(D)	BITSTRING	1	PDS2INDC	- INDICATOR BYTE
		1...		PDS2ALIS	“BIT0”- NAME IN THE FIRST FIELD IS AN ALIAS
		1...		DEALIAS	“BIT0”— ALIAS FOR PDS2ALIS
		.11.		PDS2NTTR	“BIT1+ BIT2”- NUMBER OF TTR'S IN THE USER DATA FIELD
		...1 1111		PDS2LUSR	“BIT3+ BIT4+ BIT5+ BIT6+ BIT7”- LENGTH OF USER DATA FIELD IN HALF WORDS
14	(E)	CHARACTER	1	PDS2USRD (0)	- START OF VARIABLE LENGTH USER DATA FIELD
14	(E)	CHARACTER	3	PDS2TTRT	- TTR OF FIRST BLOCK OF TEXT
17	(11)	CHARACTER	1	PDS2ZERO	- ZERO
18	(12)	CHARACTER	3	PDS2TTRN	- TTR OF NOTE LIST OR SCATTER/TRANSLATION TABLE. USED FOR MODULES IN SCATTER LOAD FORMAT OR OVERLAY STRUCTURE ONLY.
21	(15)	SIGNED	1	PDS2NL	- NUMBER OF ENTRIES IN NOTE LIST FOR MODULES IN OVERLAY STRUCTURE

Decimal Offsets	Hex Offsets	Type/Value	Len	Name (Dim)	Description
22	(16)	BITSTRING	2	PDS2ATR (0)	- TWO-BYTE MODULE ATTRIBUTE FIELD
22	(16)	BITSTRING	1	PDS2ATR1	- FIRST BYTE OF MODULE ATTRIBUTE FIELD
		1...		PDS2RENT	“BIT0”- REENTERABLE
		.1..		PDS2REUS	“BIT1”- REUSABLE
		..1.		PDS2OVLY	“BIT2”- IN OVERLAY STRUCTURE
		...1		PDS2TEST	“BIT3”- MODULE TO BE TESTED - TESTRAN
		.... 1...		PDS2LOAD	“BIT4”- ONLY LOADABLE
		.... .1..		PDS2SCTR	“BIT5”- SCATTER FORMAT
		.... ..1.		PDS2EXEC	“BIT6”- EXECUTABLE
		1		PDS21BLK	“BIT7”- IF ZERO, MODULE CONTAINS MULTIPLE RECORDS WITH AT LEAST ONE BLOCK OF TEXT. — IF ONE, MODULE CONTAINS NO RLD ITEMS AND ONLY ONE BLOCK OF TEXT.
23	(17)	BITSTRING	1	PDS2ATR2	- SECOND BYTE OF MODULE ATTRIBUTE FIELD
		1...		PDS2FLVL	“BIT0”- IF ZERO, MODULE CAN BE PROCESSED BY ALL LEVELS OF LINKAGE EDITOR. --- IF ONE, MODULE CAN BE PROCESSED ONLY BY F LEVEL OF LINKAGE EDITOR.
		.1..		PDS2ORG0	“BIT1”- LINKAGE EDITOR ASSIGNED ORIGIN OF FIRST BLOCK OF TEXT IS ZERO.
		..1.		PDS2EP0	“BIT2=- ENTRY POINT ASSIGNED BY LINKAGE EDITOR IS ZERO”
		...1		PDS2NRLD	“‘BIT3’- MODULE CONTAINS NO RLD ITEMS”
		.... 1...		PDS2NREP	“BIT4”- MODULE CANNOT BE REPROCESSED BY LINKAGE EDITOR
		.... .1..		PDS2TSTN	“BIT5”- MODULE CONTAINS TESTRAN SYMBOL CARDS
		.... ..1.		PDS2LEF	“BIT6”- MODULE CREATED BY LINKAGE EDITOR F
		1		PDS2REFR	“BIT7”- REFRESHABLE MODULE
24	(18)	SIGNED	3	PDS2STOR	- TOTAL CONTIGUOUS VIRTUAL STORAGE REQUIREMENT OF MODULE
27	(1B)	SIGNED	2	PDS2FTBL	- LENGTH OF FIRST BLOCK OF TEXT
29	(1D)	ADDRESS	3	PDS2EPA	- ENTRY POINT ADDRESS ASSOCIATED WITH MEMBER NAME OR WITH ALIAS NAME IF ALIAS INDICATOR IS ONE

Data areas

Decimal Offsets	Hex Offsets	Type/Value	Len	Name (Dim)	Description
32	(20)	ADDRESS	3	(0)	- LINKAGE EDITOR ASSIGNED ORIGIN OF FIRST BLOCK OF TEXT (OS USE OF FIELD)
32	(20)	BITSTRING	3	PDS2FTBO (0)	- FLAG BYTES (VS/1-VS/2 USE OF FIELD)
32	(20)	BITSTRING	1	PDS2FTB1	- BYTE 1 OF PDS2FTBO
		1...		PDSAOSLE	“BIT0”- MODULE HAS BEEN PROCESSED BY VS/1-VS/2 LINKAGE EDITOR
		.1..		PDS2BIG	“BIT1”- THIS MODULE REQUIRES 16M OR MORE OF VIRTUAL STORAGE.
		..1.		PDS2PAGA	“BIT2”- PAGE ALIGNMENT REQUIRED FOR LOAD MODULE
		...1		PDS2SSI	“BIT3”- SSI INFORMATION PRESENT
		.... 1...		PDSAPFLG	“BIT4”- INFORMATION IN PDSAPF IS VALID
		.... .1..		PDS2LFMT	“BIT5”- MODULE IS IN PROGRAM OBJECT FORMAT. THE PDS2FTB3 FIELD IS VALID AND CONTAINS ADDITIONAL FLAGS
33	(21)	BITSTRING	1	PDS2FTB2	- BYTE 2 OF PDS2FTBO
31	(1F)	1...	1	PDS2ALTP	"BIT0:"- ALTERNATE PRIMARY FLAG. INDICATES THE PRIMARY NAME WAS GENERATED BY THE BINDER.
		...1		PDSLRMOD	“BIT3”- PROGRAM OBJECT RESIDENCE MODE
		.... 11..		PDSAAMOD	“BIT4+ BIT5”- ALIAS ENTRY POINT ADDRESSING MODE (00=AMODE=24, 10=AMODE=31, 01=AMODE=64, 11=AMODE=ANY)
		11		PDSMAMOD	“BIT6+ BIT7”- MAIN ENTRY POINT ADDRESSING MODE
34	(22)	BITSTRING	1	PDS2RLDS (0)	NUMBER OF RLD/CONTROL RECORDS WHICH FOLLOW THE FIRST BLOCK OF TEXT
34	(22)	BITSTRING	1	PDS2FTB3	- BYTE 3 OF PDS2FTBO
		1...		PDS2NMIG	“BIT0”- THIS PROGRAM OBJECT LOAD MODULE CANNOT BE CONVERTED TO RECORD FORMAT
		.1..		PDS2PRIM	“BIT1”- FETCHOPT PRIME WAS SPECIFIED
		..1.		PDS2PACK	“BIT2” FETCHOPT PACK WAS SPECIFIED
		X'23'		PDSBCEND	“*”- END OF BASIC SECTION
		X'23'		PDSBCLN	“PDSBCEND-PDS2”- LENGTH OF BASIC SECTION

THE FOLLOWING SECTION IS FOR LOAD MODULES WITH SCATTER LOAD					
		X'23'		PDSS01	“*”- START OF SCATTER LOAD SECTION
35	(23)	SIGNED	2	PDS2SLSZ	- NUMBER OF BYTES IN SCATTER LIST
37	(25)	SIGNED	2	PDS2TTSZ	- NUMBER OF BYTES IN TRANSLATION TABLE
39	(27)	CHARACTER	2	PDS2ESDT	- IDENTIFICATION OF ESD ITEM (ESDID) OF CONTROL SECTION TO WHICH FIRST BLOCK OF TEXT BELONGS
41	(29)	CHARACTER	2	PDS2ESDC	- IDENTIFICATION OF ESD ITEM (ESDID) OF CONTROL SECTION CONTAINING ENTRY POINT
		X'2B'		PDSS01ND	“*=- END OF SCATTER LOAD SECTION”
		X'8'		PDSS01LN	“PDSS01ND-PDSS01””- LENGTH OF SCATTER LOAD SECTION

THE FOLLOWING SECTION IS FOR LOAD MODULES WITH ALIAS NAMES					
		X'2B'		PDSS02	“*”- START OF ALIAS SECTION
43	(2B)	ADDRESS	3	PDS2EPM	- ENTRY POINT FOR MEMBER NAME
		X'2B'		DEENTBK	“PDS2EPM”— ALIAS
46	(2E)	CHARACTER	8	PDS2MNM	- MEMBER NAME OF LOAD MODULE. WHEN THE FIRST FIELD (PDS2NAME) IS AN ALIAS NAME, THIS FIELD CONTAINS THE ORIGINAL NAME OF THE MEMBER EVEN AFTER THE MEMBER HAS BEEN RENAMED.
		X'36'		PDSS02ND	“*”- END OF ALIAS SECTION
		X'B'		PDSS02LN	“PDSS02ND-PDSS02”- LENGTH OF ALIAS SECTION

THE FOLLOWING SECTION IS FOR SSI INFORMATION AND IS ON A HALFWORD BOUNDARY					
54	(36)	SIGNED	2	PDSS03 (0)	- FORCE HALFWORD ALIGNMENT FOR SSI SECTION
54	(36)	CHARACTER	4	PDSSSIWD (0)	- SSI INFORMATION WORD
54	(36)	SIGNED	1	PDSCHLVL	- CHANGE LEVEL OF MEMBER
55	(37)	BITSTRING	1	PDSSSIFB	- SSI FLAG BYTE
		.1..		PDSFORCE	“BIT1”- A FORCE CONTROL CARD WAS USED WHEN EXECUTING THE IHGUAP PROGRAM
		..1.		PDSUSRCH	“BIT2”- A CHANGE WAS MADE TO MEMBER BY THE INSTALLATION, AS OPPOSED TO AN IBM-DISTRIBUTED CHANGE

		...1		PDSEMFIX	“BIT3”- SET WHEN AN EMERGENCY IBM-AUTHORIZED PROGRAM 'FIX' IS MADE, AS OPPOSED TO CHANGES THAT ARE INCLUDED IN AN IBM-DISTRIBUTED MAINTENANCE PACKAGE
		.... 1...		PDSDEPCH	“BIT4”- A CHANGE MADE TO THE MEMBER IS DEPENDENT UPON A CHANGE MADE TO SOME OTHER MEMBER IN THE SYSTEM
		.... .11.		PDSSYSGN	“BIT5+ BIT6”- FLAGS THAT INDICATE WHETHER OR NOT A CHANGE TO THE MEMBER WILL NECESSITATE A PARTIAL OR COMPLETE REGENERATION OF THE SYSTEM
		....		PDSNOSGN	“X'00'”- NOT CRITICAL FOR SYSTEM GENERATION
		.... ..1.		PDSCMSGN	“BIT6”- MAY REQUIRE COMPLETE REGENERATION
		.... .1..		PDSPTSGN	“BIT5”- MAY REQUIRE PARTIAL REGENERATION
		1		PDSIBMMB	“BIT7”- MEMBER IS SUPPLIED BY IBM
56	(38)	CHARACTER	2	PDSMBRSN	- MEMBER SERIAL NUMBER
		X'3A'		PDSS03ND	“*”- END OF SSI SECTION
		X'4'		PDSS03LN	“PDSS03ND-PDSS03”- LENGTH OF SSI SECTION

THE FOLLOWING SECTION IS FOR APF INFORMATION

		X'3A'		PDSS04	“*”- START OF APF SECTION
58	(3A)	CHARACTER	2	PDSAPF (0)	- PROGRAM AUTHORIZATION FACILITY (APF) FIELD
58	(3A)	SIGNED	1	PDSAPFCT	- LENGTH OF PROGRAM AUTHORIZATION CODE (PDSAPFAC) IN BYTES
59	(3B)	CHARACTER	1	PDSAPFAC	- PROGRAM AUTHORIZATION CODE
		X'3C'		PDSS04ND	“*”- END OF APF SECTION
		X'2'		PDSS04LN	“PDSS04ND-PDSS04”- LENGTH OF APF SECTION

THE FOLLOWING SECTION IS FOR LARGE PROGRAM OBJECTS (LPO)

		X'3C'		PDSLPO	“*”- START OF LPO SECTION
60	(3C)	SIGNED	1	PDS2LPOL	- LPO SECTION LENGTH
61	(3D)	SIGNED	4	PDS2VSTR	- VIRTUAL STORAGE REQUIREMENT FOR THIS MODULE
65	(41)	SIGNED	4	PDS2MEPA	- MAIN ENTRY POINT OFFSET
69	(45)	SIGNED	4	PDS2AEPA	- ALIAS ENTRY POINT OFFSET. ONLY VALID IF THIS IS AN DIRECTORY ENTRY IS FOR AN ALIAS

X'49'	PDSLPOND	“*”- END OF LPO SECTION
X'D'	PDSLPOLN	“PDSLPOND-PDSLPO”- LENGTH OF LPO SECTION

Cross reference

Name	Hex Offset	Hex Value
PDSAAMOD	21	C
PDSAOSLE	20	80
PDSAPF	3A	
PDSAPFAC	3B	
PDSAPFCT	3A	
PDSAPFLG	20	8
PDSBCEND	22	23
PDSBCLN	22	23
PDSCHLVL	36	
PDSCMSGN	37	2
PDSDEPCH	37	8
PDSEMFIX	37	10
PDSFORCE	37	40
PDSIBMMB	37	1
PDSLLM (alternate for PDSLPO)	3C	þ
PDSLLMLN (alternate for PDSLPON)	49	D
PDSLLMND (alternate for PDSLPOND)	49	þ
PDSLPO	3C	þ
PDSLPOL	3C	D
PDSLPOLN	49	D
PDSLPOND	49	þ
PDSLRMOD	21	10
PDSMAMOD	21	3
PDSMBRSN	38	
PDSNOSGN	37	0
PDSPTSGN	37	4
PDSSSIFB	37	
PDSSSIWD	36	
PDSSYSGN	37	6
PDSS01	22	23
PDSS01LN	29	8
PDSS01ND	29	2B
PDSS02	29	2B
PDSS02LN	2E	B
PDSS02ND	2E	36
PDSS03	36	
PDSS03LN	36	4
PDSS03ND	38	3A
PDSS04	38	3A
PDSS04LN	3B	2
PDSS04ND	3B	3C
PDSUSRCH	37	20
PDS2AEPA	45	
PDS2ALIS	D	80
PDS2ALTP	21	80
PDS2ATR	16	

Name	Hex Offset	Hex Value
PDS2ATR1	16	
PDS2ATR2	17	
PDS2BIG	20	40
PDS2CNCT	B	
PDS2EPA	1D	
PDS2EPM	2B	
PDS2EP0	17	20
PDS2ESDC	29	
PDS2ESDT	27	
PDS2EXEC	16	2
PDS2FLVL	17	80
PDS2FTBL	1B	
PDS2FTBO	20	
PDS2FTB1	20	
PDS2FTB2	21	
PDS2FTB3	22	
PDS2INDC	D	
PDS2LEF	17	2
PDS2LFMT	20	4
PDS2LIBF	C	
PDS2LJOB	C	2
PDS2LLML	3C	
PDS2LLNK	C	1
PDS2LNRM	C	0
PDS2LOAD	16	8
PDS2LUSR	D	1F
PDS2MEPA	41	
PDS2MNM	2E	
PDS2NAME	0	
PDS2NL	15	
PDS2NMIG	22	80
PDS2NREP	17	8
PDS2NRLD	17	10
PDS2NTTR	D	60
PDS2ORG0	17	40
PDS2OVLY	16	20
PDS2PACK	22	20
PDS2PAGA	20	20
PDS2PRIM	22	40
PDS2REFR	17	1
PDS2RENT	16	80
PDS2REUS	16	40
PDS2RLDS	22	
PDS2SCTR	16	4
PDS2SLSZ	23	
PDS2SSI	20	10
PDS2STOR	18	
PDS2TEST	16	10
PDS2TSTN	17	4
PDS2TTRN	12	
PDS2TTRP	8	
PDS2TTRT	E	
PDS2TTSZ	25	

Name	Hex Offset	Hex Value
PDS2USRD	E	
PDS2VSTR	3D	
PDS2ZERO	11	
PDS21BLK	16	1

PDSE directory entry returned by DESERV (SMDE data area)

This section describes the format of a PDSE entry returned by the DESERV macro. This is the SMDE data area, which is mapped by the IGWSMDE mapping macro. The formats shown here are specific to PDSE program libraries.

The system managed directory entry (SMDE) can represent either a program object in a PDSE or a load module in a PDS. The SMDE also represents both primary (member) entries and aliases. The SMDE consists of several sections that appear depending on the type of directory entry being provided:

- Primary entry for a load module: BASIC+NAME+PMAR (including PMARR)
- Alias entry for a load module: BASIC+NAME+PNAME+PMAR (including PMARR)
- Primary entry for a program object: BASIC+NAME+TOKEN+PMAR (including PMARL)
- Alias entry for a program object: BASIC+NAME+PNAME+TOKEN+PMAR (including PMARL)

The basic SMDE format is shown in Table 91.

Table 91. SMDE format

Offset	Length or Bit Pattern	Name	Description
00 (X'00')	variable	SMDE	Member directory entry (structure)
00 (X'00')	44	SMDE_BASIC	Start of basic section (character)
00 (X'00')	16	SMDE_HDR	Header (character)
00 (X'00')	8	SMDE_ID	Eyecatcher (character)
08 (X'08')	4	SMDE_LEN	Length of control block. This is the sum of the sizes of the SMDE sections and the size of the user data. (unsigned)
12 (X'0C')	1	SMDE_LVL	SMDE version number (unsigned)
	X'01'	SMDE_LVL_VAL	Constant to be used with SMDE_LVL
13 (X'0D')	3	-	Reserved
16 (X'10')	1	SMDE_LIBTYPE	Source library type. Possible values are declared below with names like SMDE_LIBTYPE_XXX. (unsigned)
	X'03'	SMDE_LIBTYPE_C370LIB	Constant to be used with SMDE_LIBTYPE
	X'02'	SMDE_LIBTYPE_HFS	Constant to be used with SMDE_LIBTYPE
	X'01'	SMDE_LIBTYPE_PDSE	Constant to be used with SMDE_LIBTYPE
	X'00'	SMDE_LIBTYPE_PDS	Constant to be used with SMDE_LIBTYPE

Table 91. SMDE format (continued)

Offset	Length or Bit Pattern	Name	Description
17 (X'11')	1	SMDE_FLAG	Flag byte (bitstring)
	1...	SMDE_FLAG_ALIAS	Entry is an alias
	.1..	SMDE_FLAG_LMOD	Member is a program
	..1.	SMDE_SYSTEM_DCB	DCB opened by the system, so DESERV returned connect tokens
	...x xxxx	*	Reserved
18 (X'12')	2	*	Reserved
20 (X'14')	5	-	Extended MLTK (character)
20 (X'14')	1	-	Reserved, must be zero
21 (X'15')	4	SMDE_MLTK	MLT and concatenation number (character)
21 (X'15')	3	SMDE_MLT	MLT of member - zero if z/OS UNIX System Services (character)
24 (X'18')	1	SMDE_CNCT	Concatenation number (unsigned)
25 (X'19')	1	SMDE_LIBF	Library flag - Z-byte (unsigned)
	X'02'	SMDE_LIBF_TASKLIB	Constant to be used with SMDE_LIBF
	X'01'	SMDE_LIBF_LINKLIB	Constant to be used with SMDE_LIBF
	X'00'	SMDE_LIBF_PRIVATE	Constant to be used with SMDE_LIBF
26 (X'1A')	2	SMDE_NAME_OFF	Name offset (signed)
28 (X'1C')	2	SMDE_USRD_LEN	User data length (signed)
28 (X'1C')	2	SMDE_PMAR_LEN	Sum of lengths of program management attribute record sections (PMAR, PMARR, PMARL) (signed)
30 (X'1E')	2	SMDE_USERD_OFF	User data offset (signed)
30 (X'1E')	2	SMDE_PMAR_OFF	Program management attribute record offset (signed)
32 (X'20')	2	SMDE_TOKEN_LEN	Token length (signed)
34 (X'22')	2	SMDE_TOKEN_OFF	Token data offset (signed)
36 (X'24')	2	SMDE_PNAME_OFF	Primary name offset, zero for nonalias SMDEs or if library type is a PDS and this is not a program. (signed)
38 (X'26')	2	SMDE_NLST_CNT	Number of note list entries that exist at beginning of user data field. Always zero for non-PDS members. (signed)
40 (X'28')	2	SMDE_C370_ATTR_OFF	Offset to C370LIB attribute word
40 (X'2A')	2	*	Reserved
44 (X'2C')	variable	SMDE_SECTIONS	Start of entry sections (character)

Table 92 on page 305 through Table 97 on page 305 shows the optional SMDE_SECTIONS, or extensions to the SMDE.

Table 92. Directory entry name section

Offset	Length or Bit Pattern	Name	Description
00 (X'00')	variable	SMDE_NAME	Name descriptor (structure)
00 (X'00')	2	SMDE_NAME_LEN	Length of entry name (signed)
2 (X'02')	variable	SMDE_NAME_VAL	Entry name (character)

Table 93. Directory entry notelist section (PDS only)

Offset	Length or Bit Pattern	Name	Description
00 (X'00')	variable	SMDE_NLST	Note list extension (structure)
00 (X'00')	4	SMDE_NLST_ENTRY	Note list entries (character)
00 (X'00')	3	SMDE_NLST_RLT	Note list record location token (character)
3 (X'03')	1	SMDE_NLST_NUM	Number of RLT described by this note list block. If 0 this is not a notelist but a data block. (unsigned)

Table 94. Directory entry token section (normal use)

Offset	Length or Bit Pattern	Name	Description
00 (X'00')	32	SMDE_TOKEN	(structure)
00 (X'00')	4	SMDE_TOKEN_CONNID	CONNECT_IDENTIFIER (unsigned)
4 (X'04')	4	SMDE_TOKEN_ITEMNO	Item number (unsigned)
08 (X'08')	24	SMDE_TOKEN_FT	File token (character)

Table 95. Directory entry token section (system DCB)

Offset	Length or Bit Pattern	Name	Description
00 (X'00')	16	*	Reserved
16 (X'10')	8	SMDE_TOKEN_BMF_CT	BMF connect token
24 (X'18')	8	SMDE_TOKEN_CDM_CT	JCDM connect token

Table 96. z/OS UNIX System Services file descriptor section

Offset	Length or Bit Pattern	Name	Description
00 (X'00')	4	SMDE_FD	(structure)
00 (X'00')	4	SMDE_FD_TOKEN	File descriptor (unsigned)

Table 97. Directory entry primary name section

Offset	Length or Bit Pattern	Name	Description
00 (X'00')	variable	SMDE_PNAME	Primary name descriptor (structure)
00 (X'00')	2	SMDE_PNAME_LEN	Length of primary name (signed)
2 (X'02')	variable	SMDE_PNAME_VAL	Primary name (character)

If the SMDE represents a directory entry for a program (either a load module or a program object) the program's attributes are defined by the PMAR structure. The PMAR is a subfield of the SMDE and its offset is defined by the field SMDE_PMAR_OFF. For the PMAR structure and mapping, see IEWPMAR in *z/OS MVS Data Areas* in z/OS Internet Library at http://www.ibm.com/systems/z/os/zos/bkserv/.

If the SMDE represents a data member of a PDS or a PDSE, the SMDE_USRD_OFF field indicates the offset into the SMDE for the user data of the directory entry.

Accessing program object class information

The initial load segment of a program object can contain a binder-generated text class that describes the other loadable classes in the program object. This descriptive class is only present in program objects stored as COMPAT=PM3 or higher and when one of the following is true:

- The module has more than one loadable text class.
- There is a loadable text class whose name does not begin with B_.

If COMPAT is specified or defaulted to PM1 or PM2, this descriptive class is suppressed with no error indication. Language Environment-enabled programs that do not use a prelinker step require the IEWBLIT control block.

The information can be accessed in one of the following two ways:

- An API can be used to retrieve the text from a stored program object.
  - The standard binder API or the fast data API can be used.
  - The class name is B_LIT and the section name is IEWBLIT.
- An external reference can be used by a program within the object to locate the text.
  - Use the symbol name IEWBLIT.
  - It must be a weak reference. For example, a WXTRN in assembler terminology. The binder cannot create the B_LIT class until after all other external symbols have been resolved. A strong reference would be flagged as an error before the class could be created.

For either of the above means of access, the calling program will see the data as a structure defined for assembler programs in the shipped IEWBLIT mapping macro. The field names in that macro match those used in Table 98 on page 306.

The structure contains names, locations, and lengths of each class in the program object. It also contains the following attribute information about each class:

- RMODE
- Boundary alignment
- Initial, deferred, or no load
- Read-only status

The structure begins with a header that specifies the number of class entries and the length of each:

Table 98. BLIT structure format

Field Name	Field Type	Offset	Length	Description
BLIT_EYE_CATCHER	Char	0	8	Constant 'IEWBLIT '

Table 98. BLIT structure format (continued)

Field Name	Field Type	Offset	Length	Description
BLIT_ LENGTH	Binary	8	4	Total structure length with entries
BLIT_VERSION	Binary	12	1	Structure version = 1
*		13	3	* reserved *
BLIT_HEADER_LENGTH	Binary	16	4	Length to CTABLE
BLIT_CENTRY_LENGTH	Binary	20	4	Length of one class entry
BLIT_CENTRY_COUNT	Signed	24	4	Number of entries in the BLIT_CTABLE
BLIT_DEFER_COUNT	Binary	28	4	Number of class entries
*		32	8	* reserved *
BLIT_LOADER_TOKEN	Char	40	8	[For loader use]
BLIT_CIE_ADDR	Pointer	48	4	Address of Import/Export table or offset in segment on DASD
BLIT_MOD_ATTR	Bit	52	4	Module attributes
BLIT_XPLINK	Bit	52.0	0.1	XPLINK linkage convention
	Bit	52.1	0.2	reserved
BLIT_LC_CLASSES	Bit	52.3	0.2	Indicates if there are loadable C classes 00=not known if there are loadable C_ classes in module 10=no loadable C_ classes in module 11=loadable C_ classes in module
*		56	8	* reserved *
BLIT_CTABLE		64		Start of class entries

Each of the class entries has the following format:

Table 99. BLIT class entry format

Field Name	Field Type	Offset	Length	Description
BLIT_CLASS_NAME	Char	0	16	Name of class, blank padded
BLIT_CLASS_LENGTH	Binary	16	4	Length of class when loaded
BLIT_CLASS_ADDR	Pointer	20	4	Address of initial load class or offset in segment on DASD or zero for deferred load class
BLIT_RMODE	Binary	24	1	00=RMODE_UNSPEC 01=RMODE_24 03=RMODE_ANY 04=RMODE_64
BLIT_ALIGN	Binary	25	1	Power of 2 class alignment in storage; for example, 3=doubleword aligned (2**3 = 8)
BLIT_LOAD_ATTR	Bit	26	1	

Table 99. BLIT class entry format (continued)

Field Name	Field Type	Offset	Length	Description
BLIT_RO	Bit	26.0	0.1	Read only if on
BLIT_NOLOAD	Bit	26.1	0.1	Always off – load classes are not listed in this table
BLIT_DEFER	Bit	26.2	0.1	Deferred load if on
BLIT_LANG_ATTR	Bit	27	1	
BLIT_PADDING	Bit	27.0	0.1	The class has 16-byte initial padding.
		28	4	* reserved *

Module Map

Obtaining needed information about entry points and other external symbols in a program object often requires many calls to the binder API. Applications can avoid this for load modules by reading the PDS member directly. They may find most of the information they need in the CESD. This option is not available for programs stored in a PDSE or in UNIX files.

The binder module map may allow the writing of single-path code to handle either format, and with a much smaller number of binder API calls than was previously necessary. In addition, if the map is part of a loaded module (MODMAP=LOAD specified when building the module) it may be located directly, as explained below.

The binder writes a module map as part of the saved module under the following circumstances:

1. There is a strong reference to IEWBMMP in one of the modules input to a bind.
2. The binder option MODMAP=LOAD or MODMAP=NOLOAD is specified.

If it exists, the module map is always in section IEWBMMP. For a module with only one text class (load module and some program objects) the map is in class B_TEXT. For program objects that already contain more than one text class, the map is built in class B_MODMAP.

If the map is in a loadable class, it may be found in the loaded module as follows:

1. IEWBMMP could be located from an address constant referencing that symbol, if the address constant is in a known location.
2. If IEWBLIT exists (built by the binder for program objects containing multiple text classes or user-defined classes), then the entry for class B_MODMAP will contain the address of that class in the loaded module.
3. The map will be at the end of the first segment (the segment containing the entry points). The last doubleword of the map contains the module map locator that can be used to find the module map. The module map locator is described under “Structure of the module map” on page 309.

If the module map is in a noload class (MODMAP=NOLOAD), the map is in section IEWBMMP and class B_MODMAP. The contents are only retrievable using the binder GETD API call (or fast data GD call) with a text buffer. The contents are also retrievable in the same way if the module map is in a load class (MODMAP=LOAD); However, the class will be B_TEXT instead of B_MODMAP if there is only one text class.

Structure of the module map

A program module map consists of a small header followed by fixed length records. These records represent segments (G), classes (C), sections (S), entry points or LD records (E) and parts (P). The fixed length records are followed by a string area containing null-terminated symbols. At the end of the module map is the module map locator. The locator is a 24 byte area that contains an 8 byte eyecatcher (IEWBMMP), followed by a doubleword containing the offset to the beginning of the module map from the beginning of the segment (or zero if the map is in a NOLOAD class).

Note: The IEWBMMP macro is shipped in the SYS1.MACLIB member IEWBMMP, which contains the mapping description for assembler programs in "Accessing program object class information" on page 306.

All sections and classes (that is, all elements) and parts resident in initial load and deferred load classes have entries in the map. Segment numbers are stored in all entries.

Unnamed sections are included in the map, but unnamed entry points and parts are omitted. Unnamed entities are those whose named are printed as $PRIV****** in the binder sysprint map. C++ function or variable names are the complete mangled name. Entry records include amode, xplink/nonxplink and code/data attributes. The table contains offsets only. There are no adcons. This choice enables the table to be stored in a NOLOAD class, if desired.

The entries are ordered by segment and then offset within segment.

Entries are organized in a hierarchy, segment/class/section/entry or segment/class/section/part. For each class, there is a class entry followed by entries for all sections that contribute to that class. These section entries correspond to ED ESD entries in the module. Each section entry is followed by entries for the entry points (corresponding to LD ESD records) and parts (corresponding to PD ESD records) in that section and class.

Example of record structure

```
Segment 1  (G)
 Class1  (C)
  Section1 (S)
   Entry1 (E)
   Entry2 (E)
  Section2  (S)
   Entry3 (E)
  Section3 (S)
 Class2 (C)
  Section2 (S)
   Entry4 (E)
   Entry5 (E)
   Entry6 (E)
  Section4  (S)
   Entry7 (E)
Segment 2 (G)
 Class3 (C)
  Section0 (S)
   Part1 (P)
   Part2 (P)
```

These records contain both the length of the name of these entities and the offset to the name from the start of the map. These names are in a string area at the end

of the map. Each symbol name in the string area is null-terminated. Segments, however, do not have an associated name.

Compile unit information

The compile unit information is that which is saved from the original object module. Such information is retained across rebinds only in program objects that are in ZOSV1R5 format or later, so it is not always available. As with symbols in the main module map, symbols stored in conjunction with the compile unit information are null-terminated

Module map format

The structure of the module map begins with a header that specifies the number of class entries and the length of each:

Table 100. Module map header

Field Name	Field Type	Offset	Length	Description
BMMP_HEADER	Char	0	24	
BMMP_EYE_CATCHER	Char	0	8	Constant 'IEWBMMP'
BMMP_LENGTH	Binary	8	4	Length including header
BMMP_VERSION	Binary	12	1	Structure version=1 or 2
*		13	3	*Reserved*
BMMP_HEADER_LENGTH	Binary	15	2	Length to BMMP_ENTRIES_
BMMP_ENTRY_LENGTH	Binary	18	2	Length of one entry
BMMP_ENTRY_COUNT	Binary	20	4	Number of entries
BMMP_ENTRIES		24		Start of entries

Each of the map entries has the following format:

Table 101. Map entry format

Field Name	Field Type	Offset	Length	Description
BMMP_ENTRY			24	
BMMP_TYPE	Char	0	1	Type of map entry (G=segment, C=class, S=section, E=entry point, P=part)
BMMP_FLAGS	Binary	1	1	
BMMP_XPLINK	Bit	1.0	1	1=XPLINK (types E/P only)
BMMP_DATA	Bit	1.1	1	1=DATA (types E/P only)
BMMP_TIME	Bit	1.2	1	1=Timestamp present in compile unit information
BMMP_AMODE	Binary	2	1	Amode (types E/P only)
BMMP_RMODE	Binary	2	1	Rmode (types G/C only)
*		3	1	*Reserved*
BMMP_SEGNUM	Binary	4	2	Segment Number
BMMP_NAME_LEN	Binary	6	2	Length of name

Table 101. Map entry format (continued)

Field Name	Field Type	Offset	Length	Description
BMMP_NAME_OFF	Binary	8	4	Offset of name from start of map
BMMP_OFF	Binary	12	4	Offset of the class/element from start of segment
BMMP_CU_INFO_OFF	Binary	16	4	Offset to compile unit info from start of map. Zero if none.
BMMP_NEXT	Binary	20	4	Offset from start of map to next entry of same type

Each entry for compile unit information has the following format:

Table 102. Compile unit information

Field Name	Field Type	Offset	Length	Description
BMMP_CU_INFO		0	20	
BMMP_ORIG_NAME_LEN	Binary	0	2	Length of data set or path name
BMMP_ORIG_MEM_LEN	Binary	2	2	Length of member name
BMMP_ORIG_NAME_OFF	Binary	4	4	Offset to data set or path name from start of the map
BMMP_ORIG_MEM_OFF	Binary	8	4	Offset to member name from start of the map
BMMP_TS_DATE	Char	12	7	Compile date yyyyddd
BMMP_TS_TIME	Char	19	9	Compile time hhmmssttt if BMMP_VERSION=2 and BMMP_TIME=1
*		19	1	*Reserved* if BMMP_VERSION = 1

Appendix F. Programming examples for binder APIs

The sample program in “Examples for binder regular API” on page 315 is intended to show how the binder application programming interface could be used by an application written in System/390* assembler language. Although the task is arbitrary, the examples shows the sequence in which the APIs can be invoked in order to accomplish the task. The program is linked non-reentrant in RMODE=24.

The program invokes the binder to include an arbitrary module (IFG0198N) from a library (ddname=LPALIB), scans through its ESD entries one section at a time, and writes the ESD entries to an output file (ddname=MYDDN). The program specifies all three types of lists on the STARTD call to show how they could be used. It also demonstrates the use of the SETO call to set an option during the dialog. Use a SYSPRINT DD to see the calls made to the binder.

The program is divided into numbered sections. Additional commentary on each of the numbered sections follows:

Section Number	Description
1	This is standard MVS entry point linkage. Register 12 is saved in the message exit specification so that the exit routine can obtain addressability to its own code and data. The BASR instruction clears the high-order byte (or bit) of register 12. This was done because the message user exit routine is entered in 31-bit addressing mode and uses register 12 as its base register. If the main program is entered in 24-bit addressing mode, the high-order byte of register 12 will contain extraneous bits unless it is cleared.
2	This logic opens the output file.
3	These specifications of the IEWBUFF macro obtain storage for the ESD and NAMES buffers and initialize them. Mapping DSECTs for the buffers are provided at the end of the program.
4	The STARTD call establishes a dialog with the binder. It is always required and sets the dialog token for use in subsequent binder calls. The dialog token must be initialized to binary zero before its first usage. The example uses all three list parameters on the STARTD call: • FILES allows us to assign a ddname to the binder's print file. Note that when using the binder regular API, any required binder files (those whose ddnames do not appear on binder control statements or as macro parameters) must have ddnames assigned in this way. • EXITS allows us to specify a message exit routine that receives control, in this case, if the message severity is greater than 0. The exit routine appears at the end of this program. • OPTIONS allow us to specify one or more options that will apply throughout the binder dialog. In this example, option TERM is set to “Y”.
5	This logic creates a binder workmod with INTENT=ACCESS. The dialog token, DTOKEN, is a required input parameter. The workmod

© Copyright IBM Corp. 1991, 2012

token, WTOKEN, is set by the binder for use on subsequent calls. The workmod token must be initialized to binary zero prior to the CREATEW call.

6 SETO is used to set the LIST option to “ALL”. Since the workmod token is provided on the macro, LIST is set at the workmod level and is valid only until the workmod is reset.

7 This step includes member IFG0198N from library LPALIB, using ddname and member name to identify the module to be included.

8 The GETN call retrieves from the workmod the names of all sections in module IFG0198N. Names are returned in the names buffer, IEWBBNL, and COUNTN is set to the number of names returned. TCOUNT is set to the total number of names in the module, regardless of the size of the buffer. For this example, the two counts should be the same. The size of the buffer is controlled by the second IEWBUFF macro in section 17, which specifies SIZE=50. This provides space for up to 50 names. Since IFG0198N has fewer than 50 sections, the GETN request reaches end of file before filling the buffer. That is why it ends with return code 4, and why TCOUNT and COUNTN are the same.

9 For each name returned in the names buffer, the program issues one GETD call to obtain the ESD data. If a large module had been processed, both the GETN and GETD calls would have been processed in a loop to accommodate the possibility that there are more names or ESD records than could be obtained in a single buffer. This example, however, assumes that all ESD entries can be returned in a single GETD call.

Assuming that any ESD entries were returned for the designated section, the program scans through the buffer and writes each ESD record to the output file designated by ddname MYDDN. It is possible, however, that the item does not exist and that the named section must be bypassed.

10 DELETEW removes the workmod from binder storage. PROTECT=YES, the default, merely indicates that the delete should fail if the workmod has been altered by the dialog. Since INTENT=ACCESS, no alteration was possible, and PROTECT=YES is ineffective.

11 ENDD ends the dialog between the program and the binder, releasing any remaining resources, closing all files, and resetting the dialog token to the null value.

12 A call to IEWBUFF (FUNC=FREEBUF) frees the NAME and ESD buffers previously obtained by IEWBUFF.

13 Once the intended task is completed, the program closes the output file and releases its buffer storage.

14 This logic represents standard MVS return linkage to the operating system.

Note that an error in the program might cause control to be passed to the ERREXIT label, where clean-up processing takes place. Three clean-up items are accomplished here:

1. If the NAME and ESD buffers were obtained previously through IEWBUFF, they are released here. IEWBUFF (FUNC=FREEBUF) is invoked to accomplish this.
2. If the binder dialog is outstanding (that is, if the STARTDialog API was invoked previously), it is ended here. Note that the workmod is deleted when the dialog is ended.
3. If the output DCB is open, it is closed and its storage is released.

15 Many macro parameters require variable length character strings. To the binder, a variable length string consists of a halfword length followed by a byte string of the designated length. The halfword length value does not include the two bytes for the length field itself.

This section illustrates the definition of some of those variable length character string constants.

16 The STARTD function call specified all three types of lists: FILES, EXITS and OPTIONS. Each of these list parameters is defined here. Although each list contains only a single entry, additional entries could have been specified by incrementing the fullword count and adding another three-item specification at the end of the list.

17 The DCB for the output file is defined for this program only. The file is not shared or used in any way by the binder.

18 This use of the IEWBUFF macro provides DSECT maps for both the ESD and names buffers. Registers 6 and 7 are dedicated as base registers for the ESD buffer header and entries, respectively. Similarly, registers 8 and 9 are dedicated to the names buffer.

Note that you must code the IEWBUFF macro within a CSECT. Also note that the VERSION parameter in IEWBUFF must match the value of the VERSION parameter in the GETN and GETD binder regular APIs.

19 The message exit routine receives control, in this example, any time a message is issued by the binder with a severity of four or greater. This routine receives control in the binder's AMODE (31). It must provide capping code to switch to AMODE(24), if necessary, then back to the binder's AMODE before returning. Refer to *z/OS MVS Programming: Assembler Services Guide* for a discussion of capping.

The exit routine is copying the message from the binder's message buffer to the output file. It could have prevented the binder from issuing the message by returning a 4 in register 15.

Examples for binder regular API

Programming example for binder regular API:

```
*********************************************************************
*                                                                   *
* LICENSED MATERIALS - PROPERTY OF IBM                              *
*                                                                   *
* 5694-A01                                                          *
*                                                                   *
* COPYRIGHT IBM CORP. 1977, 2010                                    *
```

```
*                                                                     *
* STATUS = HPM7770                                                    *
*                                                                     *
***********************************************************************
*                                                                     *
*                       SAMPLE BINDER PROGRAM                         *
*                                                                     *
*  FUNCTION: Example application which includes a module and prints   *
*      its ESD records using the Binder call interface functions      *
*      INCLUDE, GETN, and GETE.                                       *
*                                                                     *
*  PROCESSING:                                                        *
*      Broken up into these steps, and referred to by these numbers   *
*      throughout:                                                    *
*                                                                     *
*      A. Initialization:                                             *
*        1) Set up the entry point linkage                            *
*        2) Open the output dataset (MYDCB used here)                 *
*        3) Open and initialize binder ESD and NAME buffers           *
*        4) Start the Binder dialog (STARTD API call).                *
*           See '16. STARTD list specifications' in sample program    *
*           to see lists specified in this call                       *
*        5) Create a workmod with INTENT=ACCESS with CREATEW call     *
*        6) The list option is set to ALL with the SETO call          *
*                                                                     *
*      B. Main processing:                                            *
*        7) Include module with the binder INCLUDE API call. In       *
*           this example, IFG0198N from LPALIB is included.           *
*        8) With the GETN call, all section names are extracted       *
*           from the workmod.                                         *
*        9) Loop to call GETD to get ESD data for each section        *
*                                                                     *
*      C. Finishing up:                                               *
*        10) Processing is done; delete workmod with DELETEW call.    *
*        11) End dialog with ENDD call.                               *
*        12) FREEBUF (Release) our buffer storage                     *
*        13) Close the output dataset                                 *
*        14) Return to the operating system                           *
*                                                                     *
*  CONSTANTS, VARIABLES, BUFFER MAPPINGS AND MESSAGE EXIT ROUTINES:   *
*        15) Variable length string constants                         *
*        16) STARTD list specifications                               *
*        17) DCB for output file                                      *
*        18) NAMES and ESD Buffer Mappings.                           *
*        19) Message Exit Routine                                     *
*                                                                     *
***********************************************************************
***********************************************************************
* PROGRAM INITIALIZATION                                              *
***********************************************************************
***********************************************************************
*               1. Entry point linkage                                *
*                                                                     *
* This is standard MVS entry point linkage. Register 12 is saved in   *
* the message exit specification so that the exit routine can obtain  *
* addressability to its own code and data.                            *
*                                                                     *
* The BASR instruction clears the high-order byte (or bit) of         *
* register 12. This was done because the message user exit routine    *
* is entered in 31-bit addressing mode and uses register 12 as its    *
* base register. If the main program is entered in 24-bit addressing  *
* mode, the high-order byte of register 12 will contain extraneous    *
* bits unless it is cleared.                                          *
*****                                                          *****
BAGETE   CSECT
         PRINT GEN
R0       EQU   0
```

```
R1       EQU   1
R2       EQU   2
R3       EQU   3
R4       EQU   4
R5       EQU   5
R6       EQU   6
R7       EQU   7
R8       EQU   8
R9       EQU   9
R10      EQU   10
R11      EQU   11
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
         SAVE  (14,12)
         BASR  R12,0              Get 31-bit base even in 24-bit mode
         USING *,R12
         ST    R12,MESSAGE+4      Save program base for message exit
         LA    R15,SAVE
         ST    R13,SAVE+4
         ST    R15,8(,13)
         LR    R13,R15
         SPACE
         MVC   FREEBFR,ZERO       No buffers to FREEBUF yet
         MVC   CLSDCB,ZERO        No DCB to close yet
         MVC   ENDDLG,ZERO        No Dialog to end yet
*********************************************************************
*               2. Open output data set                             *
*                                                                   *
* This logic opens the output file.                                 *
*****                                                             *****
         OPEN  (MYDCB,OUTPUT)     Open output data set
         LTR   R15,R15            Successful?
         BNZ   ERREXIT            Exit if not
         MVC   CLSDCB,FOUR        We must CLOSE our DCB on exit
         SPACE
*********************************************************************
*               3. Obtain and initialize binder buffers             *
*                                                                   *
* These specifications of the IEWBUFF macro obtain storage for the  *
* ESD and NAMES buffers and initialize them. Mapping DSECTs for     *
* the buffers are provided at the end of the program.               *
*****                                                             *****
       IEWBUFF FUNC=GETBUF,TYPE=ESD
       IEWBUFF FUNC=GETBUF,TYPE=NAME
       IEWBUFF FUNC=INITBUF,TYPE=ESD
       IEWBUFF FUNC=INITBUF,TYPE=NAME
         MVC   FREEBFR,FOUR       We must FREEBUF our buffers on exit
         SPACE
*********************************************************************
*               4. Start Dialog, specifying lists                   *
*                                                                   *
* The STARTD call establishes a dialog with the binder. It is always *
* required and sets the dialog token for use in subsequent binder   *
* calls. The dialog token must be initialized to binary zero before  *
* its usage.                                                        *
*                                                                   *
* The example uses all three list parameters on the STARTD call:    *
*  - FILES allows us to assign a ddname to the binders print file.  *
*    Note that when using the binder API, any required binder files *
*    (those whose ddnames do not appear on binder control statements *
*    or as macro parameters) must have ddnames assigned in this way. *
*  - EXITS allows us to specify a message exit routine that receives *
*    control, in this case, if the message severity is greater than *
*    0. The exit routine appears at the end of this program.        *
*  - OPTIONS allow us to specify one or more options that will apply *
```

```
*    throughout the binder dialog. In this example, option TERM is   *
*    set to Y.                                                       *
*****                                                              *****
         MVC   DTOKEN,DZERO Clear dialog token
               IEWBIND FUNC=STARTD,                                    +
               RETCODE=RETCODE,                                        +
               RSNCODE=RSNCODE,                                        +
               DIALOG=DTOKEN,                                          +
               FILES=FILELIST,                                         +
               EXITS=EXITLIST,                                         +
               OPTIONS=OPTLIST,                                        +
               VERSION=4
         CLC   RSNCODE,ZERO       Check the reason code
         BNE   ERREXIT            Exit if not zero
         MVC   ENDDLG,FOUR        We must ENDDIALOG on exit
         EJECT
*********************************************************************
*               5. Create a Workmod with Intent ACCESS              *
*                                                                   *
* This logic creates a binder workmod with INTENT=ACCESS. The dialog *
* token, DTOKEN, is a required input parameter. The workmod token,   *
* WTOKEN, is set by the binder for use on subsequent calls. The      *
* workmod token must be initialized to binary zero prior to the      *
* CREATEW call.                                                     *
*****                                                              *****
         MVC   WKTOKEN,DZERO      Clear workmod token
       IEWBIND FUNC=CREATEW,                                         +
               RETCODE=RETCODE,                                        +
               RSNCODE=RSNCODE,                                        +
               WORKMOD=WKTOKEN,                                        +
               DIALOG=DTOKEN,                                          +
               INTENT=ACCESS,                                          +
               VERSION=4
         CLC   RSNCODE,ZERO       Check the reason code
         BNE   ERREXIT            Exit if not zero
         EJECT
*********************************************************************
*               6. Set the list option to ALL                       *
*                                                                   *
* SETO is used to set the LIST option to ALL. Since the workmod token*
* is provided on the macro, LIST is set at the workmod level and is  *
* valid only until the workmod is reset.                            *
*****                                                              *****
       IEWBIND FUNC=SETO,                                            +
               RETCODE=RETCODE,                                        +
               RSNCODE=RSNCODE,                                        +
               WORKMOD=WKTOKEN,                                        +
               OPTION=LIST,                                            +
               OPTVAL=ALL,                                             +
               VERSION=4
         CLC   RSNCODE,ZERO       Check the reason code
         BNE   ERREXIT            Exit if not zero
         EJECT
*********************************************************************
*                      MAIN PROGRAM                                 *
*********************************************************************
*********************************************************************
*               7. Include a module (IFG0198N)                      *
*                                                                   *
* This step includes member IFG0198N from library LPALIB, using      *
* ddname and member name to identify the module to be included.      *
*****                                                              *****
       IEWBIND FUNC=INCLUDE,                                         +
               RETCODE=RETCODE,                                        +
               RSNCODE=RSNCODE,                                        +
               WORKMOD=WKTOKEN,                                        +
               INTYPE=NAME,                                            +
```

```
               DDNAME=INCLLIB,                                          +
               MEMBER=MODNAME,                                          +
               VERSION=4
         CLC   RSNCODE,ZERO       Check the reason code
         BNE   ERREXIT            Exit if not zero
         EJECT
*************************************************************************
*              8. Get all section names from workmod                    *
*                                                                       *
* The GETN call retrieves from the workmod the names of all sections *
* in module IFG0198N. Names are returned in the names buffer,        *
* IEWBBNL, and COUNTN is set to the number of names returned. TCOUNT *
* is set to the total number of names in the module, regardless of   *
* size of the buffer. For this example, the two counts should be the *
* same. The size of the buffer is controlled by the second IEWBUFF   *
* macro in step 18, which specifies SIZE=50. This provides space     *
* for up to 50 names. Since IFG0198N has fewer than 50 sections, the *
* GETN request reaches end of file before filling the buffer. That is*
* why it ends with return code 4, and why TCOUNT and COUNTN are the  *
* same.                                                                 *
*****                                                                 *****
         MVC   CURSORN,ZERO
       IEWBIND FUNC=GETN,                                               +
               RETCODE=RETCODE,                                         +
               RSNCODE=RSNCODE,                                         +
               WORKMOD=WKTOKEN,                                         +
               AREA=IEWBBNL,                                            +
               CURSOR=CURSORN,                                          +
               COUNT=COUNTN,                                            +
               TCOUNT=TCOUNT,                                           +
               NTYPE=S,                                                 +
               VERSION=4
               CH    R15,=H'4'          RC=4 means have all names
         BE    GETNOKAY
         BH    ERREXIT             Any higher is an error
         PUT   MYDCB,MSG2MANY      RC=0: Too many sections
GETNOKAY EQU   *
         EJECT
*************************************************************************
*              9. Get ESD data for each name returned by GETN           *
*                                                                       *
* For each name returned in the names buffer, the program issues one *
* GETD call to obtain the ESD data. If a large module had been       *
* processed, both the GETN and GETD calls would have been processed  *
* in a loop to accommodate the possibility that there are more names *
* or ESD records than could be obtained in a single buffer. This     *
* example, however, assumes that all ESD entries can be returned in a*
* single GETE call.                                                     *
*                                                                       *
* Assuming that any ESD entries were returned for the designated     *
* section, the program scans through the buffer and writes each ESD  *
* record to the output file designated by ddname MYDDN. It is        *
* possible, however, that the item does not exist and that the named *
* section must be bypassed.                                             *
*****                                                                 *****
         L     R5,COUNTN          Number of sections
LOOP1    L     R3,BNL_NAME_PTR    Extract section name
         LH    R2,BNL_NAME_CHARS
         STH   R2,SECTION
         LA    R4,SECTION
         BCTR  R2,0
         EX    R2,MOVESEC
         MVC   CURSORD,ZERO       Reset cursor
       IEWBIND FUNC=GETD,                                               +
               RETCODE=RETCODE,                                         +
               RSNCODE=RSNCODE,                                         +
               WORKMOD=WKTOKEN,                                         +
```

```
               CLASS=B_ESD,                                            +
               SECTION=SECTION,                                        +
               AREA=IEWBESD,                                           +
               CURSOR=CURSORD,                                         +
               COUNT=COUNTD,                                           +
               VERSION=4
         CLC   RSNCODE,ZERO
         BE    GETDOKAY
         CLC   RETCODE,FOUR       Last buffer
         BE    GETDOKAY
         CLC   RETCODE,EIGHT      No data for item
         BNE   ERREXIT
GETDOKAY EQU   *
         L     R4,COUNTD          Number of ESD entries in buffer
         LTR   R4,R4              Skip empty section
         BZ    NEXTSECT
         LA    R7,ESDH_END        First record in ESD buffer
         SH    R7,=H'4'           Leave space for length info
         L     R0,ESDH_ENTRY_LENG
         AH    R0,=H'4'
         SLL   R0,16              Convert to LLBB form
LOOP2    DS    0H
         ST    R0,0(,R7)
         PUT   MYDCB,(R7)         Write ESD to output data set
         L     R0,0(,R7)
         A     R7,ESDH_ENTRY_LENG Move to next ESD in this section
         BCT   R4,LOOP2
NEXTSECT A     R9,BNLH_ENTRY_LENG Move to next section name
         BCT   R5,LOOP1
         SPACE
*********************************************************************
*                       END OF DATA - FINISH UP                     *
*********************************************************************
*********************************************************************
*              10. Done processing - delete workmod                 *
*                                                                   *
* DELETEW removes the workmod from binder storage. PROTECT=YES, the  *
* default, merely indicates that the delete should fail if the       *
* workmod has been altered by the dialog. Since INTENT=ACCESS, no    *
* alteration was possible, and PROTECT=YES is ineffective.           *
*****                                                          *****
       IEWBIND FUNC=DELETEW,                                         +
               RETCODE=RETCODE,                                      +
               RSNCODE=RSNCODE,                                      +
               WORKMOD=WKTOKEN,                                      +
               PROTECT=YES,                                          +
               VERSION=4
         CLC   RSNCODE,ZERO
         BNE   ERREXIT
         SPACE
*********************************************************************
*              11. End dialog                                       *
*                                                                   *
* ENDD ends the dialog between the program and the binder, releasing *
* any remaining resources, closing all files, and resetting the      *
* dialog token to the null value.                                    *
*****                                                          *****
       IEWBIND FUNC=ENDD,                                            +
               RETCODE=RETCODE,                                      +
               RSNCODE=RSNCODE,                                      +
               DIALOG=DTOKEN,                                        +
               VERSION=4
         CLC   RSNCODE,ZERO
         BNE   ERREXIT
         SPACE
*********************************************************************
*              12. FREEBUF (Release) our buffer storage             *
```

```
*****                                                                *****
FREEBUFS IEWBUFF FUNC=FREEBUF,TYPE=ESD
         IEWBUFF FUNC=FREEBUF,TYPE=NAME
*********************************************************************
*              13. Close output dataset                              *
*****                                                                *****
CLOSEDCB CLOSE (MYDCB)
      FREEPOOL MYDCB
         SPACE
*********************************************************************
*              14. Return to operating system                        *
*****                                                               *****
NORMEXIT EQU   *
         LA    R15,0              Set a reason code of zero
         B     EXIT
ERREXIT  EQU   *
         CLC   FREEBFR,FOUR       Do we need to FREEBUF our buffers?
         BNE   CHECKDLG
       IEWBUFF FUNC=FREEBUF,TYPE=ESD
       IEWBUFF FUNC=FREEBUF,TYPE=NAME
CHECKDLG CLC   ENDDLG,FOUR        Do we need to end the Dialog?
         BNE   CHECKDCB
*        Ending the dialog also deletes the workmod
       IEWBIND FUNC=ENDD,                                           +
               RETCODE=RETCODE,                                     +
               RSNCODE=RSNCODE,                                     +
               DIALOG=DTOKEN,                                       +
               PROTECT=NO,                                          +
               VERSION=4
CHECKDCB CLC   CLSDCB,FOUR        Do we need to CLOSE and FREE our DCB?
         BNE   SETRSN
         CLOSE (MYDCB)
      FREEPOOL MYDCB
SETRSN   L     R15,RSNCODE
EXIT     L     R13,SAVE+4
        RETURN (14,12),RC=(15)
*********************************************************************
*                       PROGRAM CONSTANTS                           *
*********************************************************************
DZERO    DC    2F'0'
ZERO     DC    F'0'
FOUR     DC    F'4'
EIGHT    DC    F'8'
MOVESEC  MVC   2(0,R4),0(R3)
MSG2MANY DC    Y(MSG2MZ-*,0),C'TOO MANY SECTIONS TO DISPLAY'
MSG2MZ   EQU   *
*********************************************************************
*              15. Variable length string constants                  *
*****                                                               *****
B_ESD    DC    H'5',C'B_ESD'      Class name
ALL      DC    H'3',C'ALL'        LIST option value
INCLLIB  DC    H'6',C'LPALIB'     Include library
LIST     DC    H'4',C'LIST'       LIST option keyword
MODNAME  DC    H'8',C'IFG0198N'   Member name
TERM     DC    H'4',C'TERM'       TERM option keyword
Y        DC    H'1',C'Y'          TERM option value
*********************************************************************
*              16. STARTD list specifications                        *
*****                                                               *****
FILELIST DS    0F                 ddname specifications
         DC    F'1'               Number of list entries
         DC    CL8'PRINT',F'8',A(PRINTX) Assign print file ddname
PRINTX   DC    CL8'SYSPRINT'      The ddname
         SPACE
OPTLIST  DS    0F                 Global options specifications
         DC    F'1'               Number of list entries
         DC    CL8'TERM',F'1',A(YX) Set TERM option
```

```
YX       DC    C'Y'               TERM option value
EXITLIST DS    0F                 User exit specifications
         DC    F'1'               Number of list entries
         DC    CL8'MESSAGE',F'12',A(MESSAGE) Specify MESSAGE exit
MESSAGE  DC    A(MSGEXIT)         Exit routine entry point
         DC    AL4(0)             Base address for exit routine
         DC    A(FOUR)            Take exit for severity >= 4
*********************************************************************
*                        WORKING STORAGE                            *
*********************************************************************
SAVE     DS    18F                Register save area
SAVE2    DS    18F                Another for the exit routine
SAVE13   DS    F                  Register 13 save
COUNTD   DS    F                  Number of ESD records returned
COUNTN   DS    F                  Number of section names
CURSORD  DS    F                  Cursor value for GETD call
CURSORN  DS    F                  Cursor value for GETN call
DCB@     DS    F                  DCB for output file
DTOKEN   DS    CL8                Dialog Token
RETCODE  DS    F                  General return code
RSNCODE  DS    CL4                General reason code
SECTION  DS    H,CL8              Section Name for GETD
TCOUNT   DS    F                  Total number of sections
WKTOKEN  DS    CL8                Workmod Token
MSGLEN   DS    F
MSG      DC    80C'0'             Put message buffer
FREEBFR  DS    F                  Indicator for FREEBUFing our buffers
*                                 on exit, if they were GETBUFfed.
CLSDCB   DS    F                  Indicator for closing our DCB
ENDDLG   DS    F                  Indicator for ENDDing the Dialog
*********************************************************************
*              17. DCB for output file                              *
*****                                                           *****
MYDCB    DCB   DSORG=PS,MACRF=PM,RECFM=VB,LRECL=300,DDNAME=MYDDN
*****                                                           *****
*********************************************************************
*              18. NAMES and ESD Buffer Mappings.                   *
*****                                                           *****
       IEWBUFF FUNC=MAPBUF,TYPE=ESD,SIZE=50,                          +
               HEADREG=6,ENTRYREG=7,VERSION=4
       IEWBUFF FUNC=MAPBUF,TYPE=NAME,SIZE=50,                         +
               HEADREG=8,ENTRYREG=9,VERSION=4
         LTORG
*********************************************************************
*                      MESSAGE EXIT ROUTINE                         *
*                                                                   *
*        This exit routine merely prints out a message as an example *
*        of how the print exit could be used, not how it should     *
*        be used.                                                   *
*********************************************************************
*********************************************************************
*              19. Message Exit Routine                             *
*                                                                   *
*        Note: This routine will always be entered in AMODE(31).    *
*        If AMODE(24) is required, capping code must be added.      *
*****                                                           *****
MSGEXIT  EQU   *
         SAVE  (14,12)
         L     R12,0(,R1)         Get address of user data
         L     R12,0(,R12)        Get user data(pgm base register)
         L     R4,28(,R1)         Get address of exit return code
         XC    0(4,R4),0(R4)      Set exit return code to zero
         L     R3,4(,R1)          Get address of address of msg buf
         L     R3,0(,R3)          Get address of message buffer
         LH    R1,0(,R3)          Length of the message
         LA    R0,L'MSG
         CR    R1,R0
```

```
         BNL   MSGX2
         LR    R1,R0              But limited to buffer length
MSGX2    DS    0H
         LA    R0,4(,R1)          Length+4 for QSAM
         SLL   R0,16              Convert to LLBB form
         ST    R0,MSGLEN
         BCTR  R1,0               Length-1 for Execute
         EX    R1,MOVEMSG         Put all we can in the buffer
         LA    R3,MSGLEN
         ST    R13,SAVE13         Save input save area address
         LA    R13,SAVE2          Save area for PUT
         PUT   MYDCB,(R3)         Write message to data set
         L     R13,SAVE13         Restore save area register
        RETURN (14,12)            Return to binder
*
MOVEMSG  MVC   MSG(0),2(R3) Executed above
         END BAGETE
```

Examples for binder C/C++ API

The following sample program is intended to show how the binder C/C++ programming interface could be used by an application written in C language.

1. Use binder regular APIs that include a module and print classes, sessions with CU numbers, and compile units.
2. Use fast data accesses that include a module and print classes, sessions with CU numbers, and compile units.

```
                  #pragma nomargins nosequence
/*********************************************************************/
/*                                                                   */
/* LICENSED MATERIALS - PROPERTY OF IBM                              */
/*                                                                   */
/* 5694-A01                                                          */
/*                                                                   */
/* COPYRIGHT IBM CORP. 1977, 2010                                    */
/*                                                                   */
/* STATUS = HPM7770                                                  */
/*                                                                   */
/* FUNCTION: Example program which uses the Binders C/C++ programming */
/*           interface. Here we call both the Binder API and         */
/*           fast data access with the C/C++ API.                    */
/*                                                                   */
/* PROCESSING:                                                       */
/*     A. Main program - main()                                      */
/*       1) Set up CU_TYPE array                                     */
/*       2) Calls test_binder_api()                                  */
/*       3) Calls test_fdata_api()                                   */
/*                                                                   */
/*     B. Sample Binder API program - test_binder_api()              */
/*       1) Includes a module                                        */
/*       2) Prints classes: calls print_classes()                    */
/*       3) Prints sections with CU numbers & compile units:         */
/*                          calls print_compile_units()              */
/*                                                                   */
/*     C. Sample fast data access program - test_fdata_api()         */
/*       1) Includes a module                                        */
/*       2) Prints classes: calls fd_print_classes()                 */
/*       3) Prints sections with CU numbers & compile units:         */
/*                          calls fd_print_compile_units()           */
/*                                                                   */
/*                                                                   */
/* HELPER FUNCTIONS:                                                 */
/*     A. Print classes - print_classes() & fd_print_classes()       */
/*       1) Loop on getN output to get all classes                   */
/*       2) Process all the sections retrieved:                      */
```

```
/*                          calls print_name_entry()                */
/*                                                                  */
/*     B. Print sections with CU numbers and compile units -        */
/*        print_compile_units() & fd_print_compile_units()          */
/*       1) Loop on getN output to get all sections                 */
/*       2) Process all the sections retrieved                      */
/*                          calls print_cu_name_entry()             */
/*       3) Loop on getC output to get all the compile units        */
/*       4) Process all the CUs retrieved                           */
/*                          calls print_cu_entry()                  */
/********************************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

/*******************************/
/* Binder's C/C++ Include Files */
/*******************************/
#define _IEW_TARGET_RELEASE _IEW_CURRENT_
#include <__iew_api.h>

/*******************************/
/* Prototypes                  */
/*******************************/
void test_binder_api(const char *);
void print_classes(_IEWAPIContext* _context);
void print_compile_units(_IEWAPIContext* _context);
void test_fdata_api(const char *);
void fd_print_classes(_IEWFDContext* _context);
void fd_print_compile_units(_IEWFDContext* _context);
void print_cu_entry(FILE* _file, _IEWCUIEntry* _buf, int print_cuinfo);
void print_cu_name_entry(FILE* _file, _IEWNameListEntry* _buf);
void print_name_entry(FILE* _file, _IEWNameListEntry* _buf);

char * CU_TYPE[0xFF];

/*************************************************************/
/* Name: main()                                              */
/* Input:                                                    */
/* returns:                                                  */
/* Description: Sample Binder C/C++ API program              */
/* 1. Use Binder APIs which include a module and print       */
/* classes, sections with CU numbers, and compile            */
/* units.                                                    */
/* 2. Use Fastdata APIs which include a module and print     */
/* classes, sections with CU numbers, and compile            */
/* units.                                                    */
/*************************************************************/
int main(const int argc, const char **argv)
{

  int i;
  for (i=0; i<=0xff; i++) {
    CU_TYPE[i] = "--invalid--";
  }

  CU_TYPE[0X00]="LM";
  CU_TYPE[0X01]="Gen'd by PUTD API V1";
  CU_TYPE[0X02]="Gen'd by PUTD API V2+";
  CU_TYPE[0X10]="PO1";
  CU_TYPE[0X11]="OBJ";
  CU_TYPE[0X12]="XOBJ";
  CU_TYPE[0X13]="GOFF";
  CU_TYPE[0X14]="Unknown";
  CU_TYPE[0X15]="Workmod";
  CU_TYPE[0X16]="Gen'd by binder";
```

```
  CU_TYPE[0X20]="PO2";
  CU_TYPE[0X30]="PO3";
  CU_TYPE[0X41]="PO4, z/OS 1.3 compat";
  CU_TYPE[0X42]="z/OS 1.5 compat";
  CU_TYPE[0X43]="z/OS 1.7 compat";
  CU_TYPE[0X51]="PO5, z/OS 1.8 compat";

  /* test binder api */
  test_binder_api(argv[1]);

  /* test fdata api */
  test_fdata_api(argv[1]);
  return 0;
}

/************************************************************/
/* Name: test_binder_api()                                  */
/* Input: None                                              */
/* returns: None                                            */
/* Description:                                             */
/* Example application which includes a module and          */
/* prints classes, sections with CU numbers, and            */
/* compile units.                                           */
/************************************************************/
void test_binder_api(const char *path_name) {
  _IEWAPIContext *apiCnxt, *retCnxt; /* APIContext */
  _IEWList* files_list;
  _IEWList* exits_list;
  unsigned int rc, reason;



  /* initialize api flags */
  _IEWAPIFlags apiflags= {0,0,0,0,0,0,0,0,0};

  /* files list variables */
  char* files[] = { "PRINT " };
  char* files_val[] = { "./temp" };

  /* exits list variables */
  struct _exit_address {
    void *entry_point;
    void *user_data;
    void *severity_level;
  } exits_address;

  struct _exit_address exitsA;
  struct _exit_address *exitsPtr;
  char* exits[] = { "SAVE" };
  void* exits_val[1];

  /* parms for __iew_openW() */
  char *parms="MAP=Y,XREF=Y,CASE=MIXED,TERM=Y,LIST=ALL";

  if (!__isPosixOn())
    files_val[0] = "SYSPRINT";

  exitsA.entry_point = NULL;
  exitsA.user_data = NULL;
  exitsA.severity_level = 0;
  exitsPtr = &exitsA;
  exits_val[0] = exitsPtr;

  /* create file list for __iew_openW() */
  files_list = __iew_create_list(1,files,(void **)files_val);
  if (files_list == NULL)
  {
```

```
  fprintf(stderr, " create_list error: files list is null.\n");
  return;
}

/* create exits list for __iew_openW() */
exits_list = __iew_create_list(1,exits,exits_val);
if (exits_list == NULL)
{
  fprintf(stderr," create_list error: exits_val %x.\n", exits_val[0]);
  fprintf(stderr," create_list error: *exits_val %x.\n", *exits_val);
  fprintf(stderr," create_list error: exits list is null.\n");
  return;
}

/* open workmod session, load BINDER, create api context with */
/* target release = <see above: #define _IEW_TARGET_RELEASE>, intent = access
 */
apiCnxt = __iew_openW(_IEW_TARGET_RELEASE, _IEW_ACCESS,
                      files_list, exits_list,
                      parms,
                      &rc,&reason);
if (apiCnxt == NULL)
{
  fprintf(stderr," openW error: apiCnxt is null.\n");
  fprintf(stderr," openW error: rc=%u, rs=<0X%.8X>.\n",rc,reason);
  return;
}
else
{
  fprintf(stderr," openW: rc=%u, rs=<0X%.8X>.\n",rc,reason);
}

/* set api flags for __iew_includeName() */
apiflags.__imports = 1;
apiflags.__aliases = 1;
apiflags.__attrib = 1;

/* read in program object via __iew_includeName() */
rc = __iew_includeName(apiCnxt,path_name,"",apiflags);
if (rc)
{
  fprintf(stderr," includeName error: apiCnxt is null.\n");
  fprintf(stderr," includeName error: rc=%u, rs=<0X%.8X>.\n",
          rc,__iew_get_reason_code(apiCnxt));
}
else
{
  fprintf(stderr," includeName: rc=%u, rs=<0X%.8X>.\n",
          rc,__iew_get_reason_code(apiCnxt));

  /* print all class names from workmod */
  print_classes(apiCnxt);

  /* print all compile units from workmod */
  print_compile_units(apiCnxt);
}

/* set api flags for __iew_closeW() */
apiflags.__protect = 1;

/* close workmod session, delete api context */
retCnxt = __iew_closeW(apiCnxt, apiflags, &rc, &reason);
if (retCnxt)
{
  fprintf(stderr," closeW: ERROR, context is not NULL. \n");
}
fprintf(stderr," closeW: return code = %X, rs =<0X%.8X> \n",
```

```
          rc, reason);
  return;
}


/*************************************************************/
/* Name: print_classes()                                     */
/* Input: None                                               */
/* returns: None                                             */
/* Description:                                              */
/* Print classes                                             */
/*************************************************************/
void print_classes(_IEWAPIContext* _context) {
  _IEWNameListEntry *class_name;
  unsigned int num_of_classes = 0; /* number of classes */
  int i,count;

  /* loop on __iew_getN() api to get all classes */
  while ((count = __iew_getN(_context,_IEW_CLASS, &num_of_classes,
          &class_name)) > 0)
  {
    fprintf(stderr,"print_classes: name count = %d\n",count);
    if(class_name)
    {
      /* process all CSECTs retrieved */
      for (i=0; i < count; i++)
      {
        print_name_entry(stderr,&class_name[i]);
      }
    }
    else
      fprintf(stderr,"print_classes: name buffer is NULL! \n");
  }
  if (__iew_eod(_context, (void **) &class_name, "B_BNL") != 0)
  {
    fprintf(stderr," getN: ERROR, not end of data \n");
    fprintf(stderr," getN: rc=%u, rs=<0X%.8X>.\n",
    __iew_get_return_code(_context),__iew_get_reason_code(_context));
  }
}


/*************************************************************/
/* Name: print_compile_units()                               */
/* Input: None                                               */
/* returns: None                                             */
/* Description:                                              */
/* Print sections with CU numbers, and compile units.        */
/*************************************************************/
void print_compile_units(_IEWAPIContext* _context) {
  unsigned int num_of_classes = 0; /* number of classes */
  int i,count,loopctr,print_cuinfo=0;

  /* set the first CU entry to zero, so one record of all */
  /* compile units are returned */
  int cu_val[]= {0};

  /* void *name_entry, *cu_entry;  */
  _IEWNameListEntry *sect_name;
  _IEWCUIEntry *cui_entry;

  /* loop on getN() api to get all sections */
  while ((count = __iew_getN(_context,_IEW_SECTION, &num_of_classes,
          &sect_name)) > 0)
  {
    fprintf(stderr,"print sections with CU numbers: name count = %d\n",
            count);
```

```
    if(sect_name)
    {
      /* process all CSECTs retrieved */
      for (i=0; i < count; i++)
      {
        print_cu_name_entry(stderr,&sect_name[i]);
      }
    }
    else
      fprintf(stderr,"print_compile_units: name entry is NULL! \n");
  }

  if (__iew_eod(_context, (void **) &sect_name, "B_BNL") != 0)
  {
    fprintf(stderr," getN: ERROR, not end of data \n");
    fprintf(stderr," getN: rc=%u, rs=<0X%.8X>.\n",
    __iew_get_return_code(_context),__iew_get_reason_code(_context));
    return;
  }

  /* loop on get_C() api to get all compile units */
  loopctr = 0;
  while ((count = __iew_getC(_context,cu_val,
          &cui_entry)) > 0)
  {
    loopctr++;
    if (loopctr==1 &&
        cui_entry[0].__cui_cu==0 &&
        cui_entry[0].__cui_source_cu==0 &&
        cui_entry[0].__cui_concat==0 &&
        cui_entry[0].__cui_type>=0x42)
        print_cuinfo = 1;
    fprintf(stderr,"print_compile_units: cu count = %d\n",count);
    if(cui_entry)
    {
      /* process all CUs retrieved */
      for (i=0; i < count; i++)
      {
        print_cu_entry(stderr,&cui_entry[i],print_cuinfo);
      }
    }
    else
      fprintf(stderr,"print_compile_units: CU entry is NULL! \n");
  }

  if (__iew_eod(_context, (void **) &cui_entry, "B_CUI") != 0)
  {
    fprintf(stderr," getC: ERROR, not end of data \n");
    fprintf(stderr," getC: rc=%u, rs=<0X%.8X>.\n",
    __iew_get_return_code(_context),__iew_get_reason_code(_context));
  }
}


/************************************************************/
/* Name: test_fdata_api()                                   */
/* Input: None                                              */
/* returns: None                                            */
/* Description:                                             */
/* Example application which includes a module and          */
/* prints classes, sections with CU numbers, and            */
/* compile units.                                           */
/************************************************************/
void test_fdata_api(const char *path_name) {
  _IEWFDContext *fdCnxt, *fdRetCnxt; /* FDContext */
  unsigned int rc, reason;
```

```
  /* read in program object via __iew_fd_startName() */
  /* open session, create fd context with */
  /* target release = <see above: #define _IEW_TARGET_RELEASE> */
  fdCnxt =__iew_fd_open(_IEW_TARGET_RELEASE,
                        &rc,&reason);
  if (fdCnxt == NULL)
  {
    fprintf(stderr," fd_open error: fdCnxt is null.\n");
    fprintf(stderr," fd_open error: rc=%u, rs=<0X%.8X>.\n",rc,reason);
    return;
  }
  else
  {
    fprintf(stderr," fd_open rc=%u, rs=<0X%.8X>.\n",rc,reason);
  }

  /* starting a session with a path */
  rc = __iew_fd_startName(fdCnxt,path_name,"");
  fprintf(stderr, " fd_startName rc =%d \n", rc);
  if (rc)
  {
    fprintf(stderr," fd_startName error: rc=%u, rs=<0X%.8X>.\n",
            rc,__iew_fd_get_reason_code(fdCnxt));
  }
  else
  {
    fprintf(stderr," fd_startName: rc=%u, rs=<0X%.8X>.\n",
            rc,__iew_fd_get_reason_code(fdCnxt));

    /* print all class names */
    fd_print_classes(fdCnxt);

    /* print all compile units */
    fd_print_compile_units(fdCnxt);
  }

  /* end a session */
  fdRetCnxt = __iew_fd_end(fdCnxt,&rc,&reason);
  if (fdRetCnxt)
  {
    fprintf(stderr," fd_end: ERROR, context is not NULL \n");
  }
  fprintf(stderr," fd_end: return code = %X, rs =<0X%.8X> \n",
          rc,reason);
  return;
}


/*************************************************************/
/* Name: fd_print_classes()                                  */
/* Input: None                                               */
/* returns: None                                             */
/* Description:                                              */
/* Print classes                                             */
/*************************************************************/
void fd_print_classes(_IEWFDContext* _context) {
  _IEWNameListEntry *class_name;
  int i,count;

  /* loop on __iew_fd_get_N() api to get all classes */
  while ((count = __iew_fd_getN(_context,_IEW_CLASS,
          &class_name)) > 0)
  {
    fprintf(stderr,"fd_print_classes: name count = %d\n",count);
    if(class_name)
    {
      /* process all CSECTs retrieved */
```

```
      for (i=0; i < count; i++)
      {
        print_name_entry(stderr,&class_name[i]);
      }
    }
    else
      fprintf(stderr,"fd_print_classes: name buffer is NULL! \n");
  }

  if (__iew_fd_eod(_context, (void **)&class_name, "B_BNL") != 0)
  {
    fprintf(stderr," fd_getN: ERROR, not end of data \n");
    fprintf(stderr," fd_getN: rc=%u, rs=<0X%.8X>.\n",
    __iew_fd_get_return_code(_context),
    __iew_fd_get_reason_code(_context));
  }
}



/*********************************************************/
/* Name: fd_print_compile_units()                        */
/* Input: None                                           */
/* returns: None                                         */
/* Description:                                          */
/* Print sections with CU numbers, and compile units.    */
/*********************************************************/
void fd_print_compile_units(_IEWFDContext* _context) {
  int i,count,loopctr,print_cuinfo=0;
  /* set the first CU entry to zero, so one record of all */
  /* compile units are returned */
  int cu_val[]= {0};
  _IEWNameListEntry * sect_name;
  _IEWCUIEntry * cui_entry;

  /* loop on fd_get_N() api to get all sections */
  while ((count = __iew_fd_getN(_context,_IEW_SECTION,
          &sect_name)) > 0)
  {
    fprintf(stderr,"print sections with CU numbers: name count = %d\n",
            count);
    if(sect_name)
    {
      /* process all CSECTs retrieved */
      for (i=0; i < count; i++)
      {
        print_cu_name_entry(stderr,&sect_name[i]);
      }
    }
    else
      fprintf(stderr,"fd_print_compile_units: name entry is NULL! \n");
  }

  if (__iew_fd_eod(_context, (void **)&sect_name, "B_BNL") != 0)
  {
    fprintf(stderr," fd_getN: ERROR, not end of data \n");
    fprintf(stderr," fd_getN: rc=%u, rs=<0X%.8X>.\n",
    __iew_fd_get_return_code(_context),
    __iew_fd_get_reason_code(_context));
    return;
  }

  /* loop on fd_get_C() api to get all compile units */
  loopctr=0;
  while ((count = __iew_fd_getC(_context,cu_val,
          &cui_entry)) > 0)
  {
```

```
    loopctr++;
    if (loopctr==1 &&
        cui_entry[0].__cui_cu==0 &&
        cui_entry[0].__cui_source_cu==0 &&
        cui_entry[0].__cui_concat==0 &&
        cui_entry[0].__cui_type>=0x42)
        print_cuinfo = 1;
    fprintf(stderr,"fd_print_compile_units: cu count = %d\n",count);
    if(cui_entry)
    {
      /* process all CUs retrieved */
      for (i=0; i < count; i++)
      {
        print_cu_entry(stderr,&cui_entry[i],print_cuinfo);
      }
    }
    else
      fprintf(stderr,"fd_print_compile_units: CU entry is NULL! \n");
  }

  if (__iew_fd_eod(_context, (void **)&cui_entry, "B_CUI") != 0)
  {
    fprintf(stderr," fd_getC: ERROR, not end of data \n");
    fprintf(stderr," fd_getC: rc=%u, rs=<0X%.8X>.\n",
    __iew_fd_get_return_code(_context),
    __iew_fd_get_reason_code(_context));
  }
}


/*************************************************************/
/* Name: print_cu_entry()                                    */
/* Input: file, buf                                          */
/* returns:                                                  */
/* Description: print out some CU entries                    */
/*************************************************************/
void print_cu_entry(FILE* _file, _IEWCUIEntry* _buf, int print_cuinfo) {

  fprintf(_file,
  "getC: cu=%5u, src cu=%5u, concat=%4x, seq=%3u, type(%4x)=%-25s,"
  " src type(%4x)=%-25s\n",
  _buf->__cui_cu,
  _buf->__cui_source_cu,
  _buf->__cui_concat,
  _buf->__cui_c_seq,
  _buf->__cui_type,
  CU_TYPE[_buf->__cui_type],
  _buf->__cui_c_type,
  CU_TYPE[_buf->__cui_c_type]);

  if (print_cuinfo) {
    fprintf(_file,
      " source info: ddname=%8s\n"
      "  current: dsn=%44.*s, member=%8.*s\n"
      "           path(%u)=%.*s\n"
      "     orig: dsn=%44.*s, member=%8.*s\n"
      "           path(%u)=%.*s\n",
      _buf->__cui_ddname,
      _buf->__cui_dsname_len, _buf->__cui_dsname_ptr,
      _buf->__cui_member_len, _buf->__cui_member_ptr,
      _buf->__cui_path_len,
      _buf->__cui_path_len, _buf->__cui_path_ptr,
      _buf->__cui_c_dsname_len, _buf->__cui_c_dsname_ptr,
      _buf->__cui_c_member_len, _buf->__cui_c_member_ptr,
      _buf->__cui_c_path_len,
      _buf->__cui_c_path_len, _buf->__cui_c_path_ptr);
  }
```

```
}


/************************************************************/
/* Name: print_cu_name_entry()                              */
/* Input: file, name                                        */
/* returns:                                                 */
/* Description: print out some NAME entries for CU          */
/************************************************************/
void print_cu_name_entry(FILE* _file,_IEWNameListEntry* _name) {
  int i;
  char temp_str[1025];
  temp_str[0] = '\0';

  if (((char*)_name->__bnl_name_ptr)[0] == 0) {
    sprintf(temp_str,"PC%06X",*((int*)_name->__bnl_name_ptr));
  }
  else {
    sprintf(temp_str,"%.*s",_name->__bnl_name_chars,
            (char*)_name->__bnl_name_ptr);
  }

  fprintf(_file,
          "%-16s : len - %3d, cu - %3u, count - %3d  \n",
          temp_str, _name->__bnl_name_chars,
          _name->__bnl_u1.__bnl_sect_cu,_name->__bnl_elem_count);
}


/************************************************************/
/* Name: print_name_entry()                                 */
/* Input: file, buf                                         */
/* returns:                                                 */
/* Description: print out some NAME entries                 */
/************************************************************/
void print_name_entry(FILE* _file,_IEWNameListEntry* _name) {
  int i;
  char temp_str[1025];
  temp_str[0] = '\0';

  if (((char*)_name->__bnl_name_ptr)[0] == 0) {
    sprintf(temp_str,"PC%06X",*((int*)_name->__bnl_name_ptr));
  }
  else {
    sprintf(temp_str,"%.*s",_name->__bnl_name_chars,
            (char*)_name->__bnl_name_ptr);
  }

  fprintf(_file,
          "%-16s : len - %3d, count - %3d , attributes - 0x%02X\n",
          temp_str, _name->__bnl_name_chars,
          _name->__bnl_elem_count,_name->__bnl_bind_flags);
}
```

Examples for fast data access API

Programming example for fast data access API:

```
*********************************************************************
*                                                                   *
* LICENSED MATERIALS - PROPERTY OF IBM                              *
*                                                                   *
* 5694-A01                                                          *
*                                                                   *
* COPYRIGHT IBM CORP. 1977, 2010                                    *
```

```
*                                                                     *
* STATUS = HPM7770                                                    *
*                                                                     *
***********************************************************************
*                                                                     *
*                    z/OS BINDER FAST DATA ACCESS DEMO                *
*                                                                     *
* This program shows how to use fast data access calls                *
*                                                                     *
* It expects three DD names:                                          *
* SYSIN    - contains commands that guide processing                  *
* SYSLIB   - PDSE or z/OS UNIX path that contains inspected program   *
*            objects                                                  *
* SYSPRINT - application puts all output there                        *
*                                                                     *
* All SYSIN commands except XX represent a single fast data access    *
* call.                                                               *
*                                                                     *
* SB [MEMBER] - Starts a new session using given member name          *
* SJ [MEMBER] - Starts a new session using given member name          *
*               Unlike SB, if SYSLIB specifies z/OS UNIX path, member *
*               name is appended to it                                *
* SQ [ENTRY]  - Starts a new session using given entry point name of  *
*               an already loaded program object.                     *
* GC          - Prints all compile units                              *
* GD          - Prints size of B_TEXT data in the entire module       *
* GE          - Prints all ESD entries of currently opened PO         *
* GN SECTIONS - Prints names of all sections                          *
*    CLASSES    or classes of currently opened PO                     *
* RC          - Prints last return and reason codes                   *
* EN          - Ends current session                                  *
* XX          - Stops processing. Mandatory at the very end of SYSIN  *
*                                                                     *
***********************************************************************
***********************************************************************
* Pretty names for registers                                          *
***********************************************************************
R0       EQU 0
R1       EQU 1
R2       EQU 2
R3       EQU 3
R4       EQU 4
R5       EQU 5
R6       EQU 6
R7       EQU 7
R8       EQU 8
R9       EQU 9
R10      EQU 10
R11      EQU 11
R12      EQU 12
R13      EQU 13
R14      EQU 14
R15      EQU 15
***********************************************************************
* Macro that fills the first string with blanks and then copies the   *
* second string into it                                               *
***********************************************************************
         MACRO
&NAME    STRCPY &DST,&SRC
         DS 0H
         MVI &DST,C' '
         MVC &DST+1(L'&DST-1),&DST
         MVC &DST+0(L'&SRC),&SRC
         MEND
***********************************************************************
* Entry poing linkage                                                 *
***********************************************************************
```

```
FDEMO    CSECT
FDEMO    AMODE 31                      Required to use fast data access
FDEMO    RMODE ANY
         SAVE (14,12)                  Save registers.
*
         BALR R12,0                    Establish addressability to
         USING *,R12                   code through register 12.
*
         LHI R0,DATASIZE               Get above-the-line data area
         GETMAIN RU,LV=(R0),LOC=31     and establish addressability
         LR R11,R1                     to it through
         USING DATA,R11                general purpose register 11.
*
         LHI R0,DCBSSIZE               Get below-the-line data area
         GETMAIN RU,LV=(R0),LOC=24     and establish addressability
         LR R10,R1                     to it through
         USING DCBS,R10                general purpose register 10.
*
         LM R15,R1,16(R13)             Restore registers 0, 1 and 15.
         ST R13,4(R11)                 Link our save area with
         ST R11,8(R13)                 caller's save area.
         LR R13,R11                    Put address of our save area    *
                                       into register 13.
*********************************************************************
* Clean resource acquisition status. It is represented as array of    *
* bytes each of which shows whether resource is currently obtained.   *
*********************************************************************
         XC STATUS_START(STATUS_LEN),STATUS_START
*********************************************************************
* Open SYSPRINT                                                      *
*********************************************************************
         MVC SYSPRINT(SYSPRINT_TEMPLATE_LEN),SYSPRINT_TEMPLATE         *
                                             Copy DCB below the line.
         MVC PARMLIST(OPEN_OUTPUT_LIST_LEN),OPEN_OUTPUT_LIST           *
                                             Copy parameter list.
         OPEN (SYSPRINT),MF=(E,PARMLIST),                              *
               MODE=31                       Issue open.
         TM SYSPRINT+(DCBOFLGS-IHADCB),DCBOFOPN  Check whether DCB was *
                                                 opened successfully.
         BNZ SYSPRINT_OK
SYSPRINT_XX DS 0H                       If open fails
         LHI R9,12                      set return code to 12,
         B CLEANUP                      free resources and exit.
SYSPRINT_OK DS 0H
         MVC STATUS_SYSPRINT,=XL1'1'    Mark SYSPRINT as opened.
*********************************************************************
* Print startup message                                              *
*********************************************************************
         STRCPY PRINTBUF,MSG_STARTUP   Copy message to 125-byte buffer.
         PUT SYSPRINT,PRINTBUF         Issue PUT.
*********************************************************************
* Open SYSIN                                                         *
*********************************************************************
         MVC SYSIN(SYSIN_TEMPLATE_LEN),SYSIN_TEMPLATE                  *
                                             Copy DCB below the line.
         MVC PARMLIST(OPEN_INPUT_LIST_LEN),OPEN_INPUT_LIST             *
                                             Copy parameter list.
         OPEN (SYSIN),MF=(E,PARMLIST),                                 *
               MODE=31                       Issue open.
         TM SYSIN+(DCBOFLGS-IHADCB),DCBOFOPN   Check whether DCB was   *
                                               opened successfully.
         BNZ SYSIN_OK
SYSIN_XX DS 0H                            If open fails
         STRCPY PRINTBUF,MSG_SYSIN_FAILED print error message,
         PUT SYSPRINT,PRINTBUF
         LHI R9,12                        set return code to 12,
         B CLEANUP                        free resources and exit.
```

```
SYSIN_OK DS 0H
         MVC STATUS_SYSIN,=XL1'1'         Mark SYSIN as opened.
*********************************************************************
* Open SYSLIB                                                       *
*********************************************************************
         MVC SYSLIB(SYSLIB_TEMPLATE_LEN),SYSLIB_TEMPLATE              *
                                          Copy DCB below the line.
         MVC PARMLIST(OPEN_INPUT_LIST_LEN),OPEN_INPUT_LIST            *
                                          Copy parameter list.
         OPEN (SYSLIB),MF=(E,PARMLIST),                               *
               MODE=31                    Issue open.
         TM SYSLIB+(DCBOFLGS-IHADCB),DCBOFOPN  Check whether DCB was  *
                                          opened successfully.
         BNZ SYSLIB_OK
SYSLIB_XX DS 0H                          If open fails
         STRCPY PRINTBUF,MSG_SYSLIB_FAILED print error message,
         PUT SYSPRINT,PRINTBUF
         LHI R9,12                       set return code to 12,
         B CLEANUP                       free resources and exit.
SYSLIB_OK DS 0H
         MVC STATUS_SYSLIB,=XL1'1'        Mark SYSLIB as opened.
*********************************************************************
* Load IEWBFDAT                                                     *
*********************************************************************
         LOAD EP=IEWBFDAT                   Issue LOAD.
         ST R0,IEWBFDAT                     Save entry point address.
         MVC STATUS_IEWBFDAT,=XL1'1'        Mark IEWBFDAT as loaded.
*********************************************************************
* Read from SYSLIN until XX command is encountered                  *
*********************************************************************
READLOOP DS 0H
         GET SYSIN,INPUTBUF                 Read next command.
*
         STRCPY PRINTBUF,MSG_ECHO_PREFIX    Append prefix to it
         MVC PRINTBUF+L'MSG_ECHO_PREFIX(80),INPUTBUF
         PUT SYSPRINT,PRINTBUF              and echo it to SYSPRINT.
*********************************************************************
* Dispatch command procesing to an appropriate routine              *
*********************************************************************
         CLC INPUTBUF(3),=C'SB '
         BE DO_SB
         CLC INPUTBUF(3),=C'SJ '
         BE DO_SJ
         CLC INPUTBUF(3),=C'SQ '
         BE DO_SQ
         CLC INPUTBUF(3),=C'GC '
         BE DO_GC
         CLC INPUTBUF(3),=C'GD '
         BE DO_GD
         CLC INPUTBUF(3),=C'GE '
         BE DO_GE
         CLC INPUTBUF(3),=C'GN '
         BE DO_GN
         CLC INPUTBUF(3),=C'RC '
         BE DO_RC
         CLC INPUTBUF(3),=C'EN '
         BE DO_EN
         CLC INPUTBUF(3),=C'XX '
         BE READLOOP_END
DO_INVLD DS 0H                           If it's an invalid command
         STRCPY PRINTBUF,MSG_INVALID_COMMAND
         PUT SYSPRINT,PRINTBUF           put error message
         B READLOOP                      and read the next one.
*********************************************************************
* Process SB - Start session with a BLDL identifier                 *
*********************************************************************
DO_SB    DS 0H
```

```
         MVC PGMNAME(8),INPUTBUF+3           Get member name
         XC MTOKEN,MTOKEN                    Zero out MTOKEN
         L R15,IEWBFDAT
         CALL (15),(SB,MTOKEN,SYSLIB,PGMNAME),VL,                    *
               MF=(E,PARMLIST)               Call fast data
         B READLOOP                          Process the next command.
*********************************************************************
* Process SJ - Start session with a DDNAME or PATH                  *
*********************************************************************
DO_SJ    DS 0H
         XR R0,R0                    Find length of a member:
         IC R0,=C' '                 we are searching for space
         LA R1,INPUTBUF+80           until the end of string
         LA R2,INPUTBUF+3            starting from the 4th character.
         SRST R1,R2                  Go.
         SR R1,R2                    Put length into register 1.
*
         STH 1,PGMNAME               Build vstring corresponding to a
         BCTR 1,0                    member name by copying its length
         EX 1,SJ_MVC                 and characters into PGMNAME.
*
         XC MTOKEN,MTOKEN            Zero out MTOKEN
         L R15,IEWBFDAT
         CALL (15),(SJ,MTOKEN,SYSLIB_DD_VSTRING,PGMNAME),VL,          *
               MF=(E,PARMLIST)       Call fast data
         B READLOOP                  Process the next command.
SJ_MVC   DS 0H                       Out-of-control-flow
         MVC PGMNAME+2(0),INPUTBUF+3 MVC template.
*********************************************************************
* Process SQ - Start session with a CSVQUERY token                  *
*********************************************************************
DO_SQ    DS 0H
         MVC PGMNAME(8),INPUTBUF+3           Get entry point name.
*
         CSVQUERY INEPNAME=PGMNAME,                                  *
               OUTEPTKN=EPTOKEN              Issue CSVQUERY.
         LTR R15,R15                         Check whether
         BZ CSVQUERY_OK                      CSVQUERY succeeded.
CSVQUERY_XX DS 0H                            If CSVQURY fails
         STRCPY PRINTBUF,MSG_CSVQUERY_FAILED
         PUT SYSPRINT,PRINTBUF               print error message and
         B READLOOP                          process the next command.
CSVQUERY_OK DS 0H
         L R15,IEWBFDAT
         CALL (15),(SQ,MTOKEN,EPTOKEN),VL,                           *
               MF=(E,PARMLIST)               Call fast data.
         B READLOOP                          Process the next command.
*********************************************************************
* Process GC - Get compile unit data                                *
*********************************************************************
DO_GC    DS 0H
IEWBCUI_BASE EQU R2                  Base register for CUI buffer.
CUI_BASE     EQU R3                  Base register for CUI entry.
         IEWBUFF FUNC=GETBUF,TYPE=CUI  Get memory for CUI buffer.
         IEWBUFF FUNC=INITBUF,TYPE=CUI Init CUI buffer.
*********************************************************************
* Keep calling fast data while there are more CUI entries           *
*********************************************************************
         XC CURSOR,CURSOR            Zero out cursor.
GC_LOOP  DS 0H
         L R15,IEWBFDAT
         CALL (15),(GC,MTOKEN,0,(IEWBCUI_BASE),CURSOR,COUNT),VL,     *
               MF=(E,PARMLIST)       Call fast data.
         ST R15,RETCODE              Save return
         ST R0,RSNCODE               and reason codes.
*
         CLC RETCODE,=F'4'
```

```
         BNE GC_BADRC                  We want only RETCODE=4
         CLC RSNCODE,=XL4'10800001'    and RSNCODE='10800001'X
         BE GC_OK                      (more data)
         CLC RSNCODE,=XL4'10800002'    or RSNCODE='10800002'X
         BE GC_OK                      (no more data).
GC_BADRC DS 0H                            Other codes are invalid.
         STRCPY PRINTBUF,MSG_RC           Build error message,
         LA R15,FORMAT_HEX
         CALL (15),(PRINTBUF+4,RETCODE),                              *
               MF=(E,PARMLIST)            format return
         LA R15,FORMAT_HEX
         CALL (15),(PRINTBUF+17,RSNCODE),                             *
               MF=(E,PARMLIST)            and reason codes.
         PUT SYSPRINT,PRINTBUF            Print error message.
         B FREE_CUI                       Free buffer and             *
                                          read the next command.
GC_OK    DS 0H
*********************************************************************
* Format and print entries obtained                                  *
*********************************************************************
         L R4,COUNT             Load register 4 with size of entries.
         LR R5,CUI_BASE         Save address of the first entry.
         LTR R4,R4
         BZ GC_LOOP2_END        If there are no entries, skip the loop.
GC_LOOP2 DS 0H
         STRCPY PRINTBUF,MSG_GC     Build CUI message.
         L R6,CUI_MEMBER_PTR
         LLH R7,CUI_MEMBER_LEN
         CL R7,=F'0'
         BE GC_NO_MEMBER
         BCTR R7,0
GC_LEN_OK DS 0H
         EX R7,GC_MVC               Copy name.
         PUT SYSPRINT,PRINTBUF      Print CUI message.
GC_NO_MEMBER DS 0H
         A CUI_BASE,CUIH_ENTRY_LENG Move on to the next entry.
         BCT R4,GC_LOOP2            Repeat.
GC_LOOP2_END DS 0H
         LR CUI_BASE,R5             Restore address of the first entry.
         CLC RSNCODE,=XL4'10800001'
         BE GC_LOOP                 If there are more entries, call    *
                                    fast data again.
FREE_CUI DS 0H
         IEWBUFF FUNC=FREEBUF,TYPE=CUI    Free CUI buffer.
         B READLOOP                       Read the next command.
GC_MVC   DS 0H                      Out-of-control-flow
         MVC PRINTBUF+7(0),0(R6)    MVC template.
*********************************************************************
* Process GD - Get data from any class                               *
*********************************************************************
DO_GD    DS 0H
IEWBTXT_BASE EQU R2                    Base register for TEXT buffer.
TXT_BASE     EQU R3                    Base register for TEXT entry.
         IEWBUFF FUNC=GETBUF,TYPE=TEXT   Get memory for TEXT buffer.
         IEWBUFF FUNC=INITBUF,TYPE=TEXT  Init TXT buffer.
*********************************************************************
* Keep calling fast data while there are more data                   *
*********************************************************************
         XC CURSOR,CURSOR                Zero out cursor.
         XR R5,R5                        Accumulate full size in R5.
GD_LOOP  DS 0H
         L R15,IEWBFDAT
         CALL (15),(GD,MTOKEN,B_TEXT_VSTRING,0,(IEWBTXT_BASE),CURSOR,  *
               COUNT,0),VL,MF=(E,PARMLIST) Call fast data.
         ST R15,RETCODE                  Save return
         ST R0,RSNCODE                   and reason codes.
*
```

```
         CLC RETCODE,=F'4'
         BNE GD_BADRC                    We want only RETCODE=4
         CLC RSNCODE,=XL4'10800001'      and RSNCODE='10800001'X
         BE GD_OK                        (more data)
         CLC RSNCODE,=XL4'10800002'      or RSNCODE='10800002'X
         BE GD_OK                        (no more data).
GD_BADRC DS 0H                           Other codes are invalid.
         STRCPY PRINTBUF,MSG_RC          Build error message,
         LA R15,FORMAT_HEX
         CALL (15),(PRINTBUF+4,RETCODE),                              *
               MF=(E,PARMLIST)           format return
         LA R15,FORMAT_HEX
         CALL (15),(PRINTBUF+17,RSNCODE),                             *
               MF=(E,PARMLIST)           and reason codes.
         PUT SYSPRINT,PRINTBUF           Print error message.
         B FREE_TEXT                     Free buffer and              *
                                         read the next command.
GD_OK    DS 0H
         A R5,COUNT                      Add size of data obtained.
         CLC RSNCODE,=XL4'10800001'
         BE GD_LOOP                If there are more entries, call    *
                                   fast data again.
         ST R5,COUNT               Store full size.
         STRCPY PRINTBUF,MSG_GD    Build GD message,
         LA R15,FORMAT_HEX
         CALL (15),(PRINTBUF+17,COUNT),                               *
               MF=(E,PARMLIST)     format full size.
         PUT SYSPRINT,PRINTBUF     Print GD message.
FREE_TEXT DS 0H
         IEWBUFF FUNC=FREEBUF,TYPE=TEXT   Free TXT buffer.
         B READLOOP                       Read the next command.
*********************************************************************
* Process GE - Get ESD data                                         *
*********************************************************************
DO_GE    DS 0H
IEWBESD_BASE EQU R2                  Base register for ESD buffer.
ESD_BASE     EQU R3                  Base register for ESD entry.
         IEWBUFF FUNC=GETBUF,TYPE=ESD  Get memory for ESD buffer.
         IEWBUFF FUNC=INITBUF,TYPE=ESD Init ESD buffer.
*********************************************************************
* Keep calling fast data while there are more ESD entries           *
*********************************************************************
         XC CURSOR,CURSOR             Zero out cursor.
GE_LOOP  DS 0H
         L R15,IEWBFDAT
         CALL (15),(GE,MTOKEN,0,0,(IEWBESD_BASE),CURSOR,COUNT),VL,    *
               MF=(E,PARMLIST)        Call fast data.
         ST R15,RETCODE               Save return
         ST R0,RSNCODE                and reason codes.
*
         CLC RETCODE,=F'4'
         BNE GE_BADRC                 We want only RETCODE=4
         CLC RSNCODE,=XL4'10800001'   and RSNCODE='10800001'X
         BE GE_OK                     (more data)
         CLC RSNCODE,=XL4'10800002'   or RSNCODE='10800002'X
         BE GE_OK                     (no more data).
GE_BADRC DS 0H                           Other codes are invalid.
         STRCPY PRINTBUF,MSG_RC          Build error message,
         LA R15,FORMAT_HEX
         CALL (15),(PRINTBUF+4,RETCODE),                              *
               MF=(E,PARMLIST)           format return
         LA R15,FORMAT_HEX
         CALL (15),(PRINTBUF+17,RSNCODE),                             *
               MF=(E,PARMLIST)           and reason codes.
         PUT SYSPRINT,PRINTBUF           Print error message.
         B FREE_ESD                      Free buffer and              *
                                         read the next command.
```

```
GE_OK    DS 0H
*************************************************************************
* Format and print entries obtained                                     *
*************************************************************************
         L R4,COUNT             Load register 4 with number of entries.
         LR R5,ESD_BASE         Save address of the first entry.
         LTR R4,R4
         BZ GE_LOOP2_END        If there are no entries, skip the loop.
GE_LOOP2 DS 0H
         STRCPY PRINTBUF,MSG_GE     Build ESD message,
         MVC PRINTBUF+9(2),ESD_TYPE insert entry type
         L R6,ESD_NAME_PTR
         LH R7,ESD_NAME_CHARS
         BCTR R7,0
         EX R7,GE_MVC               and name.
         PUT SYSPRINT,PRINTBUF      Print ESD message.
         A ESD_BASE,ESDH_ENTRY_LENG Move on to the next entry.
         BCT R4,GE_LOOP2            Repeat.
GE_LOOP2_END DS 0H
         LR ESD_BASE,R5             Restore address of the first entry.
         CLC RSNCODE,=XL4'10800001'
         BE GE_LOOP                 If there are more entries, call    *
                                    fast data again.
FREE_ESD DS 0H
         IEWBUFF FUNC=FREEBUF,TYPE=ESD    Free ESD buffer.
         B READLOOP                       Read the next command.
GE_MVC   DS 0H                      Out-of-control-flow
         MVC PRINTBUF+17(0),0(R6)   MVC template.
*************************************************************************
* Process GN - Get names of sections or classes                         *
*************************************************************************
DO_GN    DS 0H
         CLC INPUTBUF+3(9),=C'SECTIONS ' Determine whether we want
         BE DO_GN_S                      sections
         CLC INPUTBUF+3(8),=C'CLASSES '  or
         BE DO_GN_C                      classes.
         B READLOOP                      Otherwise read the next one.
DO_GN_S  DS 0H
         LA R8,NTYPE_SECTIONS            Request type is 'S'.
         B RESUME_GN
DO_GN_C  DS 0H
         LA R8,NTYPE_CLASSES             Request type is 'N'.
RESUME_GN DS 0H
IEWBBNL_BASE EQU R2                      Base register for BNL buffer.
BNL_BASE     EQU R3                      Base register for BNL entry.
         IEWBUFF FUNC=GETBUF,TYPE=NAME   Get memory for buffer.
         IEWBUFF FUNC=INITBUF,TYPE=NAME  Initialize buffer.
*************************************************************************
* Keep calling fast data while there are more ESD entries               *
*************************************************************************
         XC CURSOR,CURSOR                Zero out cursor.
GN_LOOP  DS 0H
         L R15,IEWBFDAT
         CALL (15),(GN,MTOKEN,(R8),(IEWBBNL_BASE),CURSOR,COUNT),VL,    *
               MF=(E,PARMLIST)           Call fast data.
         ST R15,RETCODE                  Save return
         ST R0,RSNCODE                   and reason codes.
         CLC RETCODE,=F'4'
         BNE GN_BADRC                    We want only RETCODE=4
         CLC RSNCODE,=XL4'10800001'      and RSNCODE='10800001'X
         BE GN_OK                        (more data)
         CLC RSNCODE,=XL4'10800002'      or RSNCODE='10800002'X
         BE GN_OK                        (no data)
GN_BADRC DS 0H                           Other codes are invalid.
         STRCPY PRINTBUF,MSG_RC          Build error message,
         LA R15,FORMAT_HEX
         CALL (15),(PRINTBUF+4,RETCODE),                                *
```

```
              MF=(E,PARMLIST)           format return and
         LA R15,FORMAT_HEX
         CALL (15),(PRINTBUF+17,RSNCODE),                              *
              MF=(E,PARMLIST)           reason codes.
         PUT SYSPRINT,PRINTBUF          Print error message.
         B FREE_NAME                    Free buffer and process the   *
                                        next command.
GN_OK    DS 0H
*************************************************************************
* Format and print entries obtained                                     *
*************************************************************************
         L R4,COUNT                 Get number of entries.
         LR R5,BNL_BASE             Save address of the first entry.
         LTR R4,R4                  If there are no entries,
         BZ GN_LOOP2_END            skip the loop body.
GN_LOOP2 DS 0H
         STRCPY PRINTBUF,MSG_GN     Build the message
         L R6,BNL_NAME_PTR
         LH R7,BNL_NAME_CHARS
         BCTR R7,0
         EX R7,GN_MVC               and insert the name into it.
         PUT SYSPRINT,PRINTBUF      Print the message.
         A BNL_BASE,BNLH_ENTRY_LENG Move on to the next entry.
         BCT R4,GN_LOOP2            Repeat.
GN_LOOP2_END DS 0H
         LR BNL_BASE,R5             Restore address of the first entry.
         CLC RSNCODE,=XL4'10800001' If there are more entries
         BE GN_LOOP                 then call fast data again.
FREE_NAME DS 0H
         IEWBUFF FUNC=FREEBUF,TYPE=NAME  Free buffer.
         B READLOOP                      Read the next command.
GN_MVC   DS 0H                      Out-of-control-flow
         MVC PRINTBUF+5(0),0(R6)    template MVC.
*************************************************************************
* Process RC - Get return code information                              *
*************************************************************************
DO_RC    DS 0H
         L R15,IEWBFDAT
         CALL (15),(RC,MTOKEN,RETCODE,RSNCODE),VL,                     *
               MF=(E,PARMLIST)             Call fast data.
         STRCPY PRINTBUF,MSG_RC            Build RC message,
         LA R15,FORMAT_HEX
         CALL (15),(PRINTBUF+4,RETCODE),                               *
               MF=(E,PARMLIST)             format return
         LA R15,FORMAT_HEX
         CALL (15),(PRINTBUF+17,RSNCODE),                              *
               MF=(E,PARMLIST)             and reason codes.
         PUT SYSPRINT,PRINTBUF             Print the message.
         B READLOOP                        Read the next command.
*************************************************************************
* Process EN - End session                                              *
*************************************************************************
DO_EN    DS 0H
         L R15,IEWBFDAT
         CALL (15),(EN,MTOKEN),VL,                                     *
               MF=(E,PARMLIST)             Call fast data.
         B READLOOP                        Read the next command.
*************************************************************************
* Successful end of processing                                          *
*************************************************************************
READLOOP_END DS 0H
         STRCPY PRINTBUF,MSG_ALL_OK
         PUT SYSPRINT,PRINTBUF             Print the final message.
         XR R9,R9                          Zero out return code.
*************************************************************************
* Inspect resource acquisition status and free resources                *
*************************************************************************
```

```
CLEANUP  DS 0H
         CLC STATUS_IEWBFDAT,=XL1'0'       If IEWBFDAT is loaded
         BE SKIP_IEWBFDAT
         DELETE EP=IEWBFDAT                then unload it.
SKIP_IEWBFDAT DS 0H
         CLC STATUS_SYSLIB,=XL1'0'               If SYSLIB is opened
         BE SKIP_SYSLIB
         MVC PARMLIST(CLOSE_LIST_LEN),CLOSE_LIST
         CLOSE (SYSLIB),MF=(E,PARMLIST),MODE=31  then close it.
SKIP_SYSLIB DS 0H
         CLC STATUS_SYSIN,=XL1'0'                If SYSIN is opened
         BE SKIP_SYSIN
         MVC PARMLIST(CLOSE_LIST_LEN),CLOSE_LIST
         CLOSE (SYSIN),MF=(E,PARMLIST),MODE=31   then close it.
SKIP_SYSIN DS 0H
         CLC STATUS_SYSPRINT,=XL1'0'             If SYSPRINT is opened
         BE SKIP_SYSPRINT
         MVC PARMLIST(CLOSE_LIST_LEN),CLOSE_LIST
         CLOSE (SYSPRINT),MF=(E,PARMLIST),                              *
               MODE=31                           then close it.
SKIP_SYSPRINT DS 0H
*********************************************************************
* Exit linkage                                                      *
*********************************************************************
         L R13,SAVEAREA+4           Restore caller's save area.
*
         LHI R0,DCBSSIZE
         LR R1,R10
         FREEMAIN RU,LV=(R0),A=(R1) Free below-the-line data area.
         DROP R10
*
         LHI R0,DATASIZE
         LR R1,R11
         FREEMAIN RU,LV=(R0),A=(R1) Free above-the-line data area.
         DROP R11
*
         LR R15,R9                  Put return code into register 15
         RETURN (14,12),RC=(15)     and return to caller.
*********************************************************************
* FORMAT_HEX: Format hexadecimal number                             *
* Parameter 1: pointer to 8-byte area to be filled with EBCDIC      *
*              hexadecimal representation of a number               *
* Parameter 2: pointer to a fullword number to convert              *
*********************************************************************
FORMAT_HEX DS 0H
         SAVE (14,12)           Save registers.
         L R2,0(R1)             Put buffer address into register 2.
         L R3,4(R1)
         L R3,0(R3)             Put a number into register 3.
         A R2,=F'7'             Start filling buffer from the end.
         LHI R4,8               Repeat 8 times (for each digit).
HEXLOOP  DS 0H
         LR R5,R3               Copy number info register 5.
         N R5,=XL4'0000000F'    Get the last digit.
         IC R5,HEXCHARS(R5)     Get its EBCDIC counterpart.
         STC R5,0(R2)           Put it into buffer.
         SRL R3,4               Remove the last digit.
         S R2,=F'1'             Move text buffer pointer.
         BCT R4,HEXLOOP         Repeat.
         XR R15,R15             Zero out return code.
         RETURN (14,12),RC=(15) Restore registers and return to caller.
*********************************************************************
* End of code                                                       *
*********************************************************************
         DROP 12
*********************************************************************
* Read-only initialized data                                        *
```

```
*********************************************************************
*********************************************************************
* Parameter list templates                                          *
*********************************************************************
OPEN_OUTPUT_LIST      OPEN (,(OUTPUT)),MF=L
OPEN_OUTPUT_LIST_LEN  EQU *-OPEN_OUTPUT_LIST
OPEN_INPUT_LIST       OPEN (,(INPUT)),MF=L
OPEN_INPUT_LIST_LEN   EQU *-OPEN_INPUT_LIST
CLOSE_LIST            CLOSE (),MF=L
CLOSE_LIST_LEN        EQU *-CLOSE_LIST
*********************************************************************
* DCB templates                                                     *
*********************************************************************
SYSPRINT_TEMPLATE     DCB DSORG=PS,MACRF=PM,RECFM=FB,LRECL=125,        *
               DDNAME=SYSPRINT
SYSPRINT_TEMPLATE_LEN EQU *-SYSPRINT_TEMPLATE
SYSIN_TEMPLATE        DCB DSORG=PS,MACRF=GM,RECFM=FB,LRECL=80,         *
               DDNAME=SYSIN
SYSIN_TEMPLATE_LEN    EQU *-SYSIN_TEMPLATE
SYSLIB_TEMPLATE       DCB DSORG=PO,RECFM=U,MACRF=R,                    *
               DDNAME=SYSLIB
SYSLIB_TEMPLATE_LEN   EQU *-SYSLIB_TEMPLATE
*********************************************************************
* Messages                                                          *
*********************************************************************
MSG_STARTUP           DC C'Z/OS BINDER FAST DATA API DEMO'
MSG_SYSIN_FAILED      DC C'COULD NOT OPEN SYSIN'
MSG_SYSLIB_FAILED     DC C'COULD NOT OPEN SYSLIB'
MSG_INVALID_COMMAND   DC C'INVALID COMMAND'
MSG_RC                DC C'RET=12345678 RSN=12345678'
MSG_ALL_OK            DC C'ALL OK'
MSG_MTOKEN            DC C'MTOKEN=12345678'
MSG_GE                DC C'ESD TYPE=12 NAME='
MSG_ECHO_PREFIX       DC C'* '
MSG_CSVQUERY_FAILED   DC C'CSVQUERY FAILED'
MSG_GN                DC C'NAME='
MSG_GC                DC C'MEMBER='
MSG_GD                DC C'B_TEXT DATA SIZE='
*********************************************************************
* Fast data access request codes                                    *
*********************************************************************
SB       DC C'SB',X'0001'
SJ       DC C'SJ',X'0001'
SQ       DC C'SQ',X'0001'
GC       DC C'GC',X'0001'
GD       DC C'GD',X'0001'
GE       DC C'GE',X'0001'
GN       DC C'GN',X'0001'
RC       DC C'RC',X'0001'
EN       DC C'EN',X'0001'
*********************************************************************
* GN call types                                                     *
*********************************************************************
NTYPE_SECTIONS        DC C'S'
NTYPE_CLASSES         DC C'C'
*********************************************************************
* Fast data mappings and buffer templates                           *
*********************************************************************
ESDBUF   IEWBUFF FUNC=MAPBUF,TYPE=ESD,VERSION=6,SIZE=8
CUIBUF   IEWBUFF FUNC=MAPBUF,TYPE=CUI,VERSION=6,BYTES=40960
NAMBUF   IEWBUFF FUNC=MAPBUF,TYPE=NAME,VERSION=6,SIZE=8
TXTBUF   IEWBUFF FUNC=MAPBUF,TYPE=TEXT,VERSION=6,BYTES=2048
*********************************************************************
* SYSLIB DDNAME represented as vstring                              *
*********************************************************************
SYSLIB_DD_VSTRING     DC H'6',C'SYSLIB'
*********************************************************************
```

```
* B_TEXT class name represented as vstring                          *
*********************************************************************
B_TEXT_VSTRING        DC H'6',C'B_TEXT'
*********************************************************************
* Hexadecimal characters                                            *
*********************************************************************
HEXCHARS              DC C'0123456789ABCDEF'
*********************************************************************
* Literals                                                          *
*********************************************************************
                      LTORG
*********************************************************************
* Read-write uninitialized above-the-line data                      *
*********************************************************************
DATA     DSECT
*********************************************************************
* Save area                                                         *
*********************************************************************
SAVEAREA DS 18F
*********************************************************************
* Resource acquisition status                                       *
*********************************************************************
STATUS_START    EQU *
STATUS_SYSPRINT DS XL1
STATUS_SYSIN    DS XL1
STATUS_SYSLIB   DS XL1
STATUS_IEWBFDAT DS XL1
STATUS_LEN      EQU *-STATUS_START
*********************************************************************
* Other variables                                                   *
*********************************************************************
PARMLIST DS 32F     Common area for passing parameters.
PRINTBUF DS CL125   Output buffer.
INPUTBUF DS CL80    Input buffer.
IEWBFDAT DS F       Fast data entry point.
MTOKEN   DS F       Current session identifier.
CURSOR   DS F       Fast data cursor position.
COUNT    DS F       Number of entries obtained.
PGMNAME  DS CL100   Member, path or entry point name.
EPTOKEN  DS CL8     CSVQUERY token.
RETCODE  DS F       Return code.
RSNCODE  DS F       Reason code.
DATASIZE EQU *-DATA
*********************************************************************
* Read-write uninitialized data segment (located below the line)    *
*********************************************************************
DCBS     DSECT
SYSPRINT DS 0D
         ORG SYSPRINT+SYSPRINT_TEMPLATE_LEN
SYSIN    DS 0D
         ORG SYSIN+SYSIN_TEMPLATE_LEN
SYSLIB   DS 0D
         ORG SYSLIB+SYSLIB_TEMPLATE_LEN
DCBSSIZE EQU *-DCBS
*********************************************************************
* DCB mapping                                                       *
*********************************************************************
         DCBD
         END FDEMO
```

Examples of JCL

The following JCL compiles, binds and executes the previous examples.

```
//IEWAPCOM JOB (),MSGCLASS=H,MSGLEVEL=(1,1),TIME=1,REGION=0M
//*                                                                 *
```

```
//* LICENSED MATERIALS - PROPERTY OF IBM                               *
//*                                                                    *
//* 5694-A01                                                           *
//*                                                                    *
//* COPYRIGHT IBM CORP. 1977, 2010                                     *
//*                                                                    *
//* STATUS = HPM7770                                                   *
//*                                                                    *
//* FUNCTION: This JCL compiles, binds and executes the Binder sample  *
//*           programs shipped in SAMPLIB. The Binder sample programs  *
//*           are -                                                    *
//*            IEWAPCCC - Sample Binder C/C++ API program              *
//*            IEWAPBND - Sample Binder API program                    *
//*            IEWAPFDA - Sample Binder fast data access API program   *
//*********************************************************************
// JCLLIB ORDER=(CBC.SCCNPRC)
// SET SAMPLIB=SYS1.SAMPLIB
// SET SAMPUNIT=SYSALLDA
//*
//* Compile, bind and execute IEWAPCCC - C API example
//*
//IEWAPCCC EXEC PROC=EDCXCB,
// CPARM='RENT,SE(/usr/include),LIST(DD:SYSPRINT)',
// INFILE=&SAMPLIB(IEWAPCCC),
// TUNIT=&SAMPUNIT
//BIND.SYSLMOD  DD DSN=&&EXECPCCC(IEWAPCCC),
//            SPACE=(TRK,(1,1,1)),DSNTYPE=LIBRARY,DISP=(NEW,PASS),
//            UNIT=&SAMPUNIT
//BIND.SYSIN DD *
  INCLUDE '/usr/lib/iewbnddx.x'
/*
//*
//GOWAPCCC EXEC PGM=*.IEWAPCCC.BIND.SYSLMOD,
// PARM='ENVAR("LIBPATH=/usr/lib"),MSGFILE(MYSTDERR)//bin/sh'
//SYSPRINT DD SYSOUT=*
//MYSTDERR DD SYSOUT=*
//*
//* Compile, bind and execute IEWAPBND - Binder API example
//*
//IEWAPBND EXEC PGM=ASMA90
//SYSPRINT DD SYSOUT=*
//SYSLIN   DD DSN=&&OBJAPBND,
//            DISP=(NEW,PASS),SPACE=(TRK,(3,3)),
//            DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200),
//            UNIT=&SAMPUNIT
//SYSLIB   DD DSN=SYS1.MACLIB,DISP=SHR
//SYSIN    DD DSN=&SAMPLIB(IEWAPBND),DISP=SHR
//*
//BINDPBND EXEC PGM=IEWBLINK
//SYSPRINT DD SYSOUT=*
//SYSLMOD  DD DSN=&&EXECPBND(IEWAPBND),
//            SPACE=(TRK,(1,1,1)),DISP=(NEW,PASS),UNIT=&SAMPUNIT
//SYSLIN   DD DSN=&&OBJAPBND,DISP=(OLD,DELETE)
//*
//GOWAPBND EXEC PGM=*.BINDPBND.SYSLMOD
//MYDDN    DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//LPALIB   DD DSN=SYS1.LPALIB,DISP=SHR
//*
//* Compile, bind and execute IEWAPFDA - Fast Data API example
//*
//IEWAPFDA EXEC PGM=ASMA90
//SYSPRINT DD SYSOUT=*
//SYSLIN   DD DSN=&&OBJAPFDA,
//            DISP=(NEW,PASS),SPACE=(TRK,(3,3)),
//            DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200),
//            UNIT=&SAMPUNIT
```

```
//SYSLIB   DD DSN=SYS1.MACLIB,DISP=SHR
//SYSIN    DD DSN=&SAMPLIB(IEWAPFDA),DISP=SHR
//*
//BINDPFDA EXEC PGM=IEWBLINK
//SYSPRINT DD SYSOUT=*
//SYSLMOD  DD DSN=&&EXECPFDA(IEWAPFDA),
//            SPACE=(TRK,(1,1,1)),DSNTYPE=LIBRARY,DISP=(NEW,PASS),
//            UNIT=&SAMPUNIT
//SYSLIN   DD DSN=&&OBJAPFDA,DISP=(OLD,DELETE)
//*
//GOWAPFDA EXEC PGM=*.BINDPFDA.SYSLMOD
//SYSPRINT DD SYSOUT=*
//SYSLIB   DD DSN=&&EXECPFDA,DISP=(SHR,PASS)
//SYSIN DD *
SB IEWAPFDA
RC
GN SECTIONS
RC
GN CLASSES
RC
GE
RC
GC
RC
GD
RC
EN
XX
//*
```

Appendix G. Using the transport utility (IEWTPORT)

The transport utility (IEWTPORT) is a program management service with very specific and limited function. It obtains (via the binder) a program object from a PDSE and converts it into a “transportable program file” in a sequential (nonexecutable) format. It also reconstructs the program object from a transportable program file and stores it back into a PDSE (through the binder).

Note: You would only use this utility if you wanted to access the program data in a program object on a system where program management was NOT available. It should be emphasized that the purpose of IEWTPORT is very limited. Its use is discouraged. It is not intended to replace or provide an alternative to IEBCOPY, which is the appropriate utility for copying load modules and program objects. It is simply a service for allowing the transport of program data for other reasons than binding or loading. This utility will not be enhanced further and might be withdrawn at some future date.

You create a transportable copy of the program object using IEWTPORT, then send the transportable copy to the system without program management services. A program on the target system can access the transportable copy using QSAM. Section “Logical structure of a transportable file” on page 350 contains a description of the transportable file format. Macro IEWTFMT contains the mappings for the transportable file records.

If you want to load, bind, or execute a transportable program, you must first recreate the program object by executing IEWTPORT on a system with program management services installed. No programming interfaces exist to perform any of these operations on transportable programs.

IEWTPORT does not support load modules, nor does it support program objects in overlay format.

IEWTPORT does not support PO4 format program objects.

IEWTPORT does not support the following dynamic allocation (DYNALLOC or SVC 99) options for any data sets: S99TIOEX(XTIOT), S99ACUCB (NOCAPTURE), and S99DSABA (DSAB above the line).

Executing IEWTPORT

IEWTPORT is an executable program. You can use a batch job to invoke the IEWTPORT utility. The following is the JCL syntax:

```
//PROGA     EXEC   PGM=IEWTPORT
//SYSUT1    DD     DSN=input.dset[(member name)],DISP=SHR
//SYSUT2    DD     DSN=output.dset,DISP=(NEW,CATLG,DELETE),...
//SYSPRINT  DD     SYSOUT=*
```

input.dset[(member name)]
: The name of a PDSE program library (with or without a member specification) or the name of a sequential data set containing a transportable program.

output.dset
: The name of a PDSE program library or the name of a sequential data set containing a transportable program.

© Copyright IBM Corp. 1991, 2012

Defining the data sets

SYSPRINT is a required data set. It contains IEWTPORT user messages, binder messages, and processing information.

SYSUT1 defines the required input data set, and SYSUT2 defines the required output data set. If SYSUT1 defines a PDSE program library, SYSUT2 must define a sequential data set (sequential and extended format data sets are both supported). If SYSUT1 defines a sequential data set, then SYSUT2 must define a PDSE program library. The SYSUT1 member name specification can be a primary name or an alias. If an alias name is specified, IEWTPORT will convert the corresponding primary name.

If SYSUT1 defines a PDSE program library and SYSUT2 defines a sequential data set, IEWTPORT builds one or more transportable programs. If you include a member name, IEWTPORT builds a transportable program for that member only. If you specify a program library without a member name, IEWTPORT converts the entire program library.

If SYSUT1 defines a sequential data set containing a transportable program, and SYSUT2 defines a PDSE program library, IEWTPORT recreates the corresponding program object with its original member name, aliases, and attributes. If you specify a member name in the SYSUT2 statement and that member name already exists in the output library, IEWTPORT replaces the old member name. If you specify a member name that does not exist in the output library, IEWTPORT fails with no messages.

IEWTPORT writes the sequential data set containing transportable programs with a logical record length of 4096 bytes and a record format of variable block (LRECL=4096 and RECFM=VB). If you specify the JCL parameters BLKSIZE, LRECL, or RECFM, they are ignored.

Allocating space for the SYSUTn data sets

The disk or tape space requirement for the new data set defined by the SYSUT2 DD statement is approximately equal to that of the input data. If you are converting a single program object to its transportable format, allocate the same number of bytes as currently used by the PDSE member. If you are converting an entire library, allocate the same number of bytes as currently used by the PDSE program library. When converting a transportable file back to one or more program objects, the disk space requirement is approximately equal to the size of the transportable file.

Transporting selected members

You can create a sequential data set that contains transportable programs for all members of a PDSE program library. In this case, you can recreate the PDSE program library from the sequential data set.

If you want to create transportable programs for only some of the members of a program library, you can copy all of the selected program objects to a new PDSE program library and then convert the entire new PDSE to a single transportable file. Alternatively, you can invoke IEWTPORT once for each member and create a transportable file for each.

If you attempt to convert an empty library to its transportable format, IEWTPORT creates an empty transportable file.

Sample IEWTPORT invocations

This topic includes the following JCL examples for invoking IEWTPORT.

- "Convert a program object to a transportable program"
- "Convert an entire program library"
- "Convert a transportable program to a single program object"

Convert a program object to a transportable program

```
//CONV1     EXEC  PGM=IEWTPORT
//SYSUT1    DD    DSN=JUNE.CALCS(CALC1),DISP=SHR
//SYSUT2    DD    DSN=JUNE.CALC1.TRANS,DISP=(NEW,CATLG,DELETE),
//          SPACE=(TRK,(2,1)),UNIT=SYSDA
//SYSPRINT  DD    SYSOUT=*
```

The data set JUNE.CALCS is a PDSE program library. IEWTPORT converts the program object CALC1 and writes it to a sequential data set called JUNE.CALC1.TRANS. The messages generated by IEWTPORT are written to the SYSPRINT data set.

Convert an entire program library

```
//CONVALL   EXEC  PGM=IEWTPORT
//SYSUT1    DD    DSN=YEARLY.CALCS,DISP=SHR
//SYSUT2    DD    DSN=ALL.CALCS.TRANS,DISP=OLD
//SYSPRINT  DD    SYSOUT=*
```

The data set YEARLY.CALCS is a PDSE program library with several members in it. ALL.CALCS.TRANS has been previously allocated as a sequential data set. IEWTPORT converts the entire YEARLY.CALCS program library to a single sequential file, ALL.CALCS.TRANS. At completion, ALL.CALCS.TRANS consists of a series of transportable programs.

Convert a transportable program to a single program object

```
//RECREAT1  EXEC   PGM=IEWTPORT
//SYSUT1    DD     DSN=FEB.CALC2.TRANS,DISP=SHR
//SYSUT2    DD     DSN=FEB.CALCS,DISP=OLD
//SYSPRINT  DD     SYSOUT=*
//IEWTRACE  DD     SYSOUT=*
```

The data set FEB.CALC2.TRANS is a sequential data set containing one transportable program. FEB.CALCS is the name of a PDSE program library. IEWTPORT recreates the program object from the transportable program read in from the sequential data set, FEB.CALC2.TRANS. The program object is restored with its original name, alias names, and attributes.

Messages, errors, and return codes

This topic summarizes the IEWTPORT messages, errors, and return codes.

Messages and codes

The IEWTPORT messages are numbered in the range of 3000-3100, and the application identifier is IEW. The severity codes and the letter suffix to the message

number are the same as those for the binder. See *z/OS MVS System Messages, Vol 8 (IEF-IGD)* for the message text and the explanations.

Errors

When creating a transportable file, a severe error might cause the last written transportable program to be unusable.

If the time and date are not available due to an error, they are not recorded in the header. IEWTPORT does not generate a return code or message for this condition.

If you specify the BLKSIZE, LRECL, RECFM, PARM, or output member name in your JCL, they are ignored without any warning message or return code.

If you specify an incorrect member name for either input or output, IEWTPORT fails with no messages.

Return codes

Table 103. IEWTPORT return codes

Return Code	Description
00	Informational: IEWTPORT finished processing normally.
04	Warning: An exceptional condition has been detected, but should have no adverse effect on the conversion process. Processing continues with no user action required. The output should be verified to assure completeness and correctness of the conversion.
08	Error: IEWTPORT has detected an error in the user data such as an incorrect program object. IEWTPORT has taken corrective action. A message is issued and processing continues, if appropriate. The integrity of any output data is not assured.
12	Severe: IEWTPORT encountered an error that renders the output unusable. The error might be caused by conditions such as an improper execution environment or an I/O error. IEWTPORT has taken corrective action. The standard action is to issue a message and terminate processing.
16	Terminating: Processing ends immediately. This return code might be issued due to the lack of minimum requirements for execution of IEWTPORT, such as a missing ddname or the lack of storage. It might be returned after the failure of a system service or macro invoked by IEWTPORT. IEWTPORT has taken corrective action. The standard action is to issue a message and terminate processing. The output, if any, should be considered unusable.

Logical structure of a transportable file

Programming Interface information

A transportable file is a sequential file containing one or more transportable programs (converted program objects). It consists of a header, one or more transportable programs, and a trailer. The header indicates the beginning of the file, and the trailer indicates the end of the file. A transportable program contains the program object data. The overall file structure is shown in Figure 50 on page 351. Each of the partitions indicated in the figure represents a logical record. The content and structure of each type of the logical records is described in the remainder of

this section.

End of Programming Interface information

Transportable file logical record layout

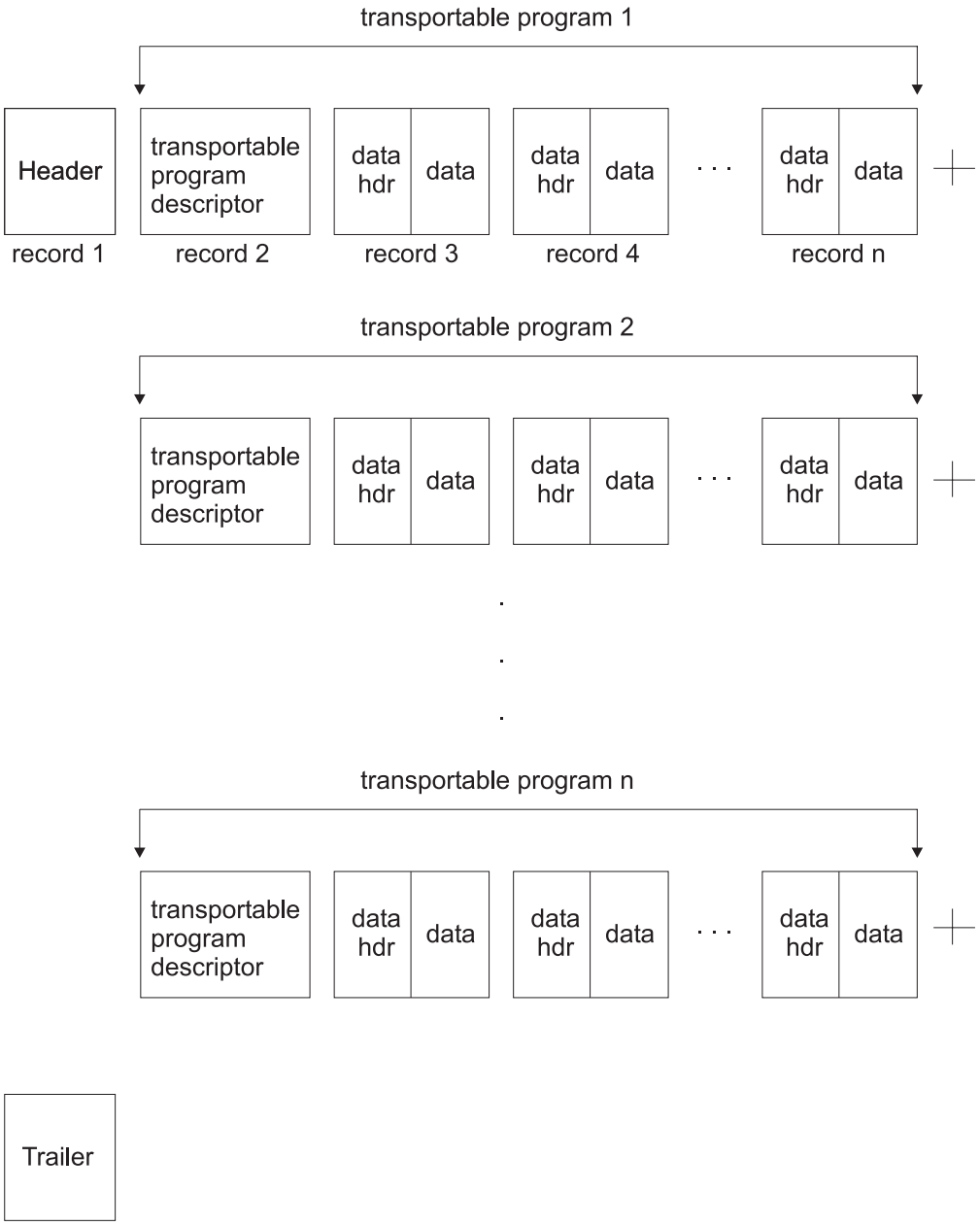


Figure 50. Transportable file structure

Mapping macro IEWTFMT

Programming Interface information

An assembler language mapping macro, IEWTFMT, is available that contains mappings of the logical records of a transportable file. Table 104 on page 352 lists the names of the DSECT maps and the record mapped by each DSECT.

You can use this macro in your program, provided you assemble the program with the Assembler H or higher program product and provided the IEWTFMT macro is available in a macro library.

Table 104. Transportable program record

DSECT name	Mapped record
TFMT_THHEAD	Header
TFMT_TTTRAIL	Trailer
TFMT_TPDESC	Descriptor
TFMT_TDALIAS	Alias data record
TFMT_TDATTRIBS	Attributes data record
TFMT_TDITEM	Item data record

End of Programming Interface information

Header

Programming Interface information

The header contains information such as the time of day and date when IEWTPORT created the transportable file and the data set name of the PDSE program library. It is mapped by TFMT_THHEAD, which is shown in Table 105.

Table 105. Transportable file header

TFMT_THHEAD	DSECT		Maps the header record
TFMT_THEYE	DS	CL8	Transportable file identifier. Contains string 'IEWTPORT'
TFMT_THLVL	DS	FL1	Current level number: 1
TFMT_THRS1	DS	CL3	Reserved
TFMT_THLEN	DS	FL4	Record length including the varying length of the data set name
TFMT_THDS_OFF	DS	FL2	Data set name offset with respect to the beginning of TFMT_THHEAD
TFMT_THRS2	DS	FL2	Reserved
TFMT_THDATE	DS	CL10	Date as MM/DD/YYYY
TFMT_THTIME	DS	CL8	Time as HH:MM:SS
TFMT_THRS3	DS	FL2	Reserved
TFMT_THDSNAME	DSECT		Maps the data set name
TFMT_THDSN_LEN	DS	FL2	Length of the data set name
TFMT_THDSN_VAL	DS	CL44	Data set name, varying, with a maximum length of 44 bytes

End of Programming Interface information

Trailer

Programming Interface information

The trailer indicates the end of the transportable program file. It is mapped by TFMT_TTTRAIL, which is shown in Table 106.

Table 106. Transportable file trailer

TFMT_TTTRAIL	DSECT		Maps trailer record
TFMT_TTEYE	DS	CL8	Trailer record identifier. Contains string 'IEWTPTRL'
TFMT_TTLVL	DS	FL1	Current level number: 1
TFMT_TTRS1	DS	FL3	Reserved
TFMT_TTLEN	DS	FL4	Trailer record length
TFMT_TTNUM	DS	FL4	Number of transportable programs in this transportable file.

End of Programming Interface information

Transportable program

Programming Interface information

Each transportable program contains a descriptor and a body. A transportable program descriptor (TP descriptor) indicates the beginning of a transportable program and contains the primary name. The body contains the data that is required to rebuild the program object. The overall structure of a transportable program is shown in Figure 51. The TP descriptor record is mapped by TFMT_TPDESC, which is shown in Table 107.

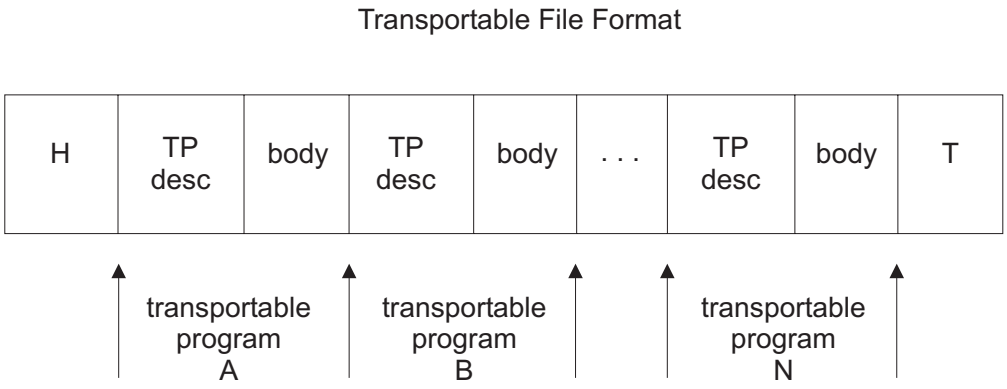


Figure 51. Transportable program structure

Table 107. Transportable program descriptor map

TFMT_TPDESC	DSECT		Maps the descriptor record
TFMT_TPEYE	DS	CL8	Transportable program descriptor identifier. Contains string 'IEWTPDSC'
TFMT_TPLVL	DS	FL1	Level number: 1
TFMT_TPPOV	DS	FL1	Program object version

Table 107. Transportable program descriptor map (continued)

TFMT_TPRS1	DS	CL2	Reserved
TFMT_TPLEN	DS	FL4	Record length including the varying member name
TFMT_TPNAME_OFF	DS	FL2	Offset to the primary (member) name relative to the beginning of TFMT_TPDESC
TFMT_TPRS2	DS	FL2	Reserved
TFMT_TPMEMNAM	DSECT		Maps primary name
TFMT_TPMEM_LEN	DS	FL2	Length of the primary name
TFMT_TPMEM_VAL	DS	CL8	Primary (member) name, varying, with a maximum length of eight bytes.

Body

The body consists of a series of data records. Each data record has a data header that describes the data that follows it. The data type is classified as either ALIAS, ITEM, or ATTRIBUTES. The data header is mapped by one of three different data sections of the IEWTFMT mapping macro, depending upon the data type.

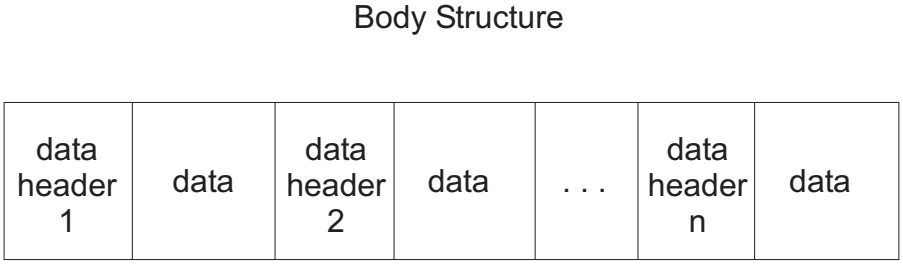


Figure 52. Transportable program body structure

In the body of a transportable program, the first records are alias records, followed by the attributes record, and then the item records. There can be multiple alias and item records but only one attributes record.

The maximum size of a data record cannot exceed the sequential data set block size of 4KB. Items larger than 4KB span several blocks.

ALIAS data type: An *ALIAS* data type indicates that the data is an alias name. For the alias data type, the alias name and AMODE are stored after the data header. Figure 53 depicts the format of the alias data type. The alias data header is mapped by the TFMT_TDALIAS section of the IEWTFMT macro and shown in Table 108 on page 355.

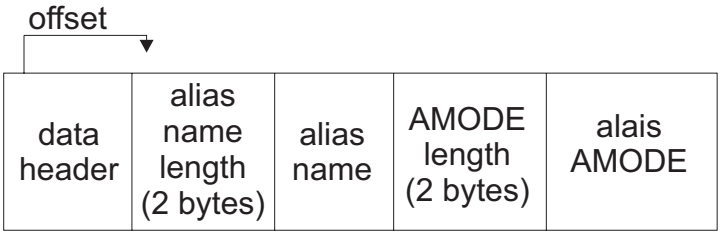


Figure 53. Alias data record

The alias data header is mapped by the TFMT_TDALIAS section of the IEWTFMT macro shown here.

Table 108. Transportable program alias data header

TFMT_TDALIAS	DSECT		Maps the alias data header
TFMT_TDALIAS_EYE	DS	CL8	Data record identifier. Contains string 'IEWTLIAS'.
TFMT_TDALIAS_LVL	DS	FL1	Level number: 1
TFMT_TDALIAS_RS1	DS	CL3	Reserved
TFMT_TDALIAS_LEN	DS	FL4	Length of the record including the varying data
TFMT_TDALIAS_EPOFF	DS	FL4	Alias entry point offset. The offset is relative to the beginning of the program object that contains this alias.
TFMT_TDALIAS_DATOFF	DS	FL2	Offset to the alias data relative to the beginning of the alias data record
TFMT_TDALIAS_RS2	DS	FL2	Reserved

Note: The varying alias data is stored immediately after the TFMT_TDALIAS data header. The alias data linear format is:

```
Alias name length  +  Alias name  + Alias AMODE length + Alias AMODE
   (2 bytes)          (varying)       (2 bytes)          (varying)
```

ATTRIBUTES data type: An **ATTRIBUTES** data type indicates that the data is an array of attributes. The attributes data header is mapped by the TFMT_TDATTRIBS section of the IEWTFMT macro. The map is described in Table 109 on page 356.

The attributes are the program attributes that have been set by the binder as options. Specifically, they are: AC, AMODE, PAGE, DC, EDIT, EP, EXEC, FETCHOPT, OL, REUS, RMODE, SCTR, SSI, and TEST. The program attributes have the same names and values as the binder options, as described in Chapter 7, “Setting options with the regular binder API,” on page 173, except for the following:

- The EP value is an entry point offset.
- The EXEC attribute indicates if the program is executable. The values are YES or NO.
- The PAGE attribute indicates if page alignment is performed for this program. The values are YES or NO.

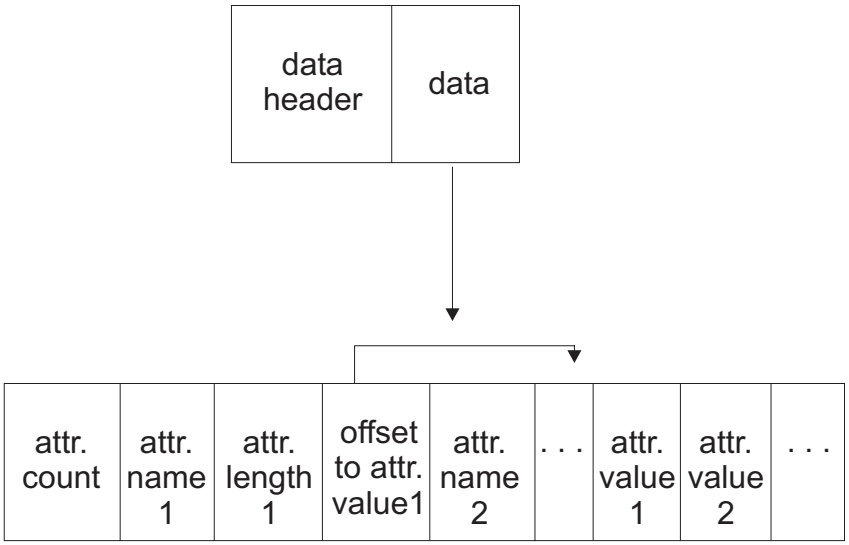


Figure 54. Attributes Data Record

Table 109. Transportable program attributes data header

TFMT_TDATTRIBS	DSECT		Maps the attributes data header
TFMT_TDATTRIBS_EYE	DS	CL8	Data record identifier. Contains string 'IEWTATTR'.
TFMT_TDATTRIBS_LVL	DS	FL1	Level number: 1
TFMT_TDATTRIBS_RS1	DS	CL3	Reserved
TFMT_TDATTRIBS_LEN	DS	FL4	Attribute data record length including the varying data
TFMT_TDATTRIBS_DLEN	DS	FL4	Length of the varying attributes data
TFMT_TDATTRIBS_OFF	DS	FL2	Offset to the attributes varying data. The offset is relative to the beginning of TFMT_TDATTRIBS.
TFMT_TDATTIBS_RS2	DS	FL2	Reserved

Note: The attributes data is stored immediately after the attributes data header. The varying attributes data has this format:

```
ATTRIBUTES_COUNT(how many in array)                (4 bytes)
ATTRIBUTES_ARRAY
  ATTRIBUTE_NAME                                   (8 bytes)
  ATTRIBUTE_LENGTH                                 (2 bytes)
  ATTRIBUTE_VALUE_OFFSET                           (2 bytes)
  The offset is relative to the beginning of this
  attributes array.
  The attribute value has a length of ATTRIBUTE_LENGTH.

ACTUAL VARYING VALUES (saved contiguously to previous structure)
```

The linear format of this data is:

```
ATTRIBUTES_COUNT (N) +
ATTRIBUTE_NAME(1) + ATTRIBUTE_LENGTH(1) + OFFSET(1) +
ATTRIBUTE_NAME(2) + ATTRIBUTE_LENGTH(2) + OFFSET(2) + ...
ATTRIBUTE_NAME(N) + ATTRIBUTE_LENGTH(N) + OFFSET(N) +
ACTUAL ATTRIBUTE VALUE(1) + ACTUAL ATTRIBUTE VALUE(2) + ...
ACTUAL ATTRIBUTE VALUE(N).
```

ITEM data type: An **ITEM** data type indicates that the data is a data item identified by a class name. For the item data type, the class name is stored after the data header. The item data itself (that is, the class of data for all sections of a program object) follows the class name.

Figure 55 on page 357 shows the structure of the item data record. The data header is mapped by the TFMT_TDITEM section in the IEWTFMT macro and is shown in Table 110 on page 357.

The data item that follows the class field is mapped by one of the mapping macros described in Appendix D, “Binder API buffer formats,” on page 239. The applicable map is determined by the class name. For example, if the class name is “B_RLD”, the corresponding map is structure IEWBRLD described in Figure 30 on page 249.

The meaning of some fields in the header structures described in Appendix D, “Binder API buffer formats,” on page 239 is changed to fit the context of a record rather than an in-storage buffer. These changes apply to the header fields listed below. Replace “xxx” with a data class identifier (ESD, RLD, IDB, ...).

xxxH_BUFFER_LENG
: Actual length of the item data including header

xxxH_ENTRY_COUNT
: Actual number of entries in the record

xxxH_NAMEPTR_ADJ
: Negative adjustment factor to add to name pointers. The pointers then become offsets from the first byte of the buffer header (for example, IEWBESD, or IEWBRLD). In the context of the item record, the offsets are relative to the beginning of the item data. See Figure 55 on page 357.

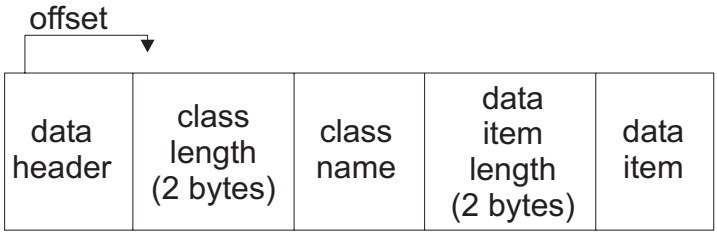


Figure 55. Item data record

Table 110. Transportable program item data header

TFMT_TDITEM	DSECT		Maps item data header
TFMT_TDITEM_EYE	DS	CL8	Data record identifier. Contains string 'IEWTITEM'
TFMT_TDITEM_LVL	DS	FL1	Level number: 1
TFMT_TDITEM_BUFF_LVL	DS	FL1	Version of the data buffer that follows this header
TFMT_TDITEM_RS1	DS	CL2	Reserved
TFMT_TDITEM_LEN	DS	FL4	Data record length including the varying data
TFMT_TDITEM_CSR	DS	FL4	Cursor number. Indicates the relationship of data in this data record relative to the previous item data record
TFMT_TDITEM_CNT	DS	FL4	Indicates how many bytes/records of binder data are saved after this header.
TFMT_TDITEM_OFF	DS	FL2	Offset to the item data relative to the beginning of TFMT_TDITEM
TFMT_TDITEM_RS2	DS	CL2	Reserved

Note: The varying item data is stored immediately after the item data header. The varying item data has the format:

```
Class name length  +  Class name  + Item data length + Item data
   (2 bytes)           (varying)       (2 bytes)       (varying)
```

End of Programming Interface information

Appendix H. Establishing installation defaults

Note: This procedure involves making changes to an MVS component and modifying an authorized library. It can only be performed by your system programmer, who can provide you with a list of the defaults that have been established for your installation.

Default values for binder options can be customized to the needs of your installation by replacing a data-only load module in SYS1.LINKLIB. CSECT IEWBODEF in load module IEWBLINK can be used to specify default values for one or more options. The module is initially set to the null string but can be modified and configured to contain binder option default settings in the same form they would be specified in the JCL PARM field, without enclosing apostrophes or parentheses. A halfword length field preceding the string must be set to the current length of the string.

The following example shows how you can set the defaults for binder options FETCHOPT, LET and MSGLEVEL:

```
IEWBODEF  CSECT
          DC     AL2(ENDPARM-PARMS)
PARMS     EQU    *
          DC     C'COMPRESS=YES,'
          DC     C'FETCHOPT(NOPACK,NOPRIME),'
          DC     C'LET(4),'
          DC     C'MSGLEVEL(4)'
ENDPARM   EQU    *
          END
```

Any options that can be set in the JCL PARM field can be defaulted in this manner. Notice the setting of the length and the comma delimiters between options. Once this module has been assembled it must be linked into module IEWBLINK, using either the linkage editor or binder, replacing the empty module.

It is recommended that only binder processing options, (for example, COMPAT, LINECT, LIST, MAP, and MSGLEVEL), be established for your installation. Module attributes, such as RMODE, REUS and OVLY, tend to vary from module to module, and changing the defaults for these attributes can result in unwanted conflicts. Note that utilities, such as IEBCOPY, invoke the binder to perform part of their processing, and any defaults established with this procedure can affect those utilities as well.

Installations planning to migrate from PDS load modules to PDSE or UNIX program objects may want to consider an installation default of COMPRESS=YES. This might result in as much as a 20 percent to 25 percent reduction in DASD space requirements for programs. However, program objects bound with COMPRESS=YES cannot be rebound or inspected (by AMBLIST or debuggers, for example) on any system older than z/OS version 1 release 7.

The installation default can be overridden for calls to the binder, including those through the c89 and ld UNIX commands, TSO LINK, and the binder Application Programming Interface.

Note: The installation default also applies in some cases to IEBCOPY and the UNIX cp command. You cannot override the default for those programs.

© Copyright IBM Corp. 1991, 2012

By default, if a parameter list does not have the VL bit on, the following message is written to the job log:

```
IEW2090W IEWBFDAT CALLED WITHOUT VL BIT ON.
```

Meanwhile, a return error code of 12 and a reason code of 10800023 are displayed.

An installation can override the requirement that the interface must use the VL format. The high order bit is set on the last parameter pointer in the VL format. To override the requirement, provide a module named IEWBQHOB that is available to fast data using an undirected LOAD without DCB. The IEWBQHOB module is provided in the LINKLIB or LPALIB concatenation, though a STEPLIB can be used if the override is wanted for only particular cases.

As a migration aid, an installation can modify the behavior by creating the module IEWBQHOB beginning with the following parameters:

SKIP No message and continue as if the VL bit is set

WARN
Write message IEW2090W and continue as if the VL bit is set

For example:

```
IEWBQHOB CSECT
         DC   C'WARN'
         END  IEWBQHOB
```

Appendix I. Accessibility

Publications for this product are offered in Adobe Portable Document Format (PDF) and should be compliant with accessibility standards. If you experience difficulties when using PDF files, you may view the information through the z/OS Internet Library web site or the z/OS Information Center. If you continue to experience problems, send an email to mhvrcfs@us.ibm.com or write to:

IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size.

Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Vol I* for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

z/OS information

z/OS information is accessible using screen readers with the Library Server versions of z/OS books in the Internet library at:

`http://www.ibm.com/systems/z/os/zos/bkserv/`

© Copyright IBM Corp. 1991, 2012

Notices

This information was developed for products and services offered in the U.S.A. or elsewhere.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

© Copyright IBM Corp. 1991, 2012

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

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Site Counsel
IBM Corporation
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

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

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Programming interface information

This book primarily documents information that is NOT intended to be used as Programming Interfaces of z/OS.

This book also documents intended Programming Interfaces that allow the customer to write programs to obtain the services of z/OS. This information is identified where it occurs, either by an introductory statement to a chapter or section or by the following marking:

Programming Interface information

End of Programming Interface information

Policy for unsupported hardware

Various z/OS elements, such as DFSMS, HCD, JES2, JES3, and MVS™, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Index

A

access intent
  specifying 42, 80
  valid function calls 10
accessibility 361
  contact IBM 361
  features 361
  screen readers 363
ADDA (add alias function)
  parameter list 30
  return and reason codes 30
  syntax 28
ALIAS data type 354
alias specifying 29
aligning sections
  4 KB boundary
    with ALIGNT function call 30
ALIGNT (align text function)
  parameter list 32
  return and reason codes 31
  syntax 30
alternate entry point
  specifying 29
  specifying AMODE 29
ALTERW (alter workmod function)
  parameter list 35
  return and reason codes 35
  syntax 32
AMASPZAP IDR data 248
AMODE specifying 29
API 1
  buffer header 241
  call version 8
  invoking from high-level languages 20
  retrieve text 306
  setting invocation environment 19
  setting options 173
  writing user exit routines 92
API (application programming interface)
  using IEWBIND macro 23
  using IEWBUFF macro 13
  writing user exit routines 177
API buffer formats 239
  AMASPZAP IDR data 248
  binder IDR data 245
  binder name list 251, 259, 275, 285, 287
  content extent list 251
  ESD entry 243, 253, 268, 278
  internal symbol table 250
  language processor IDR data 246, 286
  module map 260
  PARTINIT entry 274
  PMAR entry 277
  RLD entry 249, 257, 272
  text data buffer 250
  user IDR data 247
assistive technologies 361
ATTACH macro invoking binder 169
AUTOC (API function)
  syntax 36
AUTOCall (API function)
  parameter list 37
  return and reason codes 37
automatic library call
  BINDW function 38
  specifying option 86

B

B_DESCR 3
B_ESD 3
B_IDRB 3
B_IDRL 3
B_IDRU 3
B_IDRZ 3
B_IMPEXP 3
B_LIT 3
B_MAP 3
B_PARTINIT 3
B_PRV 2
B_RLD 3
B_SYM 3
B_TEXT 2
B_TEXT24 3
B_TEXT31 3
behavioral attributes data field 212
bind intent
  specifying 42, 80
  valid function calls 10
binder (program management binder)
  invoking
    from a program 169
    with the API 1
  program names 20, 169
binder application programming interface
  description 1
  example 313
binder dialog
  description 1
  starting 91
  summary 8
  terminating 46
binder function call
  overview 8
  sequence 8
  summary 28
binder invocation environment 19
binder options
  advanced facilities 173
  defaults 359
  installation 359
  setting up 359
  with the API 173
    using SETO function 88
    using STARTD function 91

© Copyright IBM Corp. 1991, 2012

binder processing intent
  associating with workmod 10
  description 2
binder user exit
  message exit routine 178
  save exit reason codes 179
  save exit routine 178
  specifying with STARTD function 92
BINDW (bind workmod function)
  parameter list 41
  resolving external references 38
  return and reason codes 39
  syntax 37
BLIT
  class entries 307
  description 307
  structure 306
buffer
  header contents 241
  header format 241
  values 242
buffer operations
  allocating 15
  initializing 16
  mapping 16
  releasing 14
byte-oriented data 218

C

C/C++ API 131
CALLERID option
  purpose 174
CALLIB option
  purpose 173
  specifying 88, 93
CESD record format 196
changing external symbols 33
class
  attributes 5
  description 2
  names 2, 205
class data, obtaining 109
class identifier
  B_DESCR 3
  B_ESD 3
  B_IDRB 3
  B_IDRL 3
  B_IDRU 3
  B_IDRZ 3
  B_IMPEXP 3
  B_LIT 3
  B_MAP 3
  B_PARTINIT 3
  B_PRV 2
  B_RLD 3
  B_SYM 3
  B_TEXT 2
  B_TEXT24 3
  B_TEXT31 3
coding the IEWBIND macro 23
common area
  aligning
    with ALIGNT function call 31
  changing 33
  data classes 2
  deleting 33
  expanding 33
  inserting 68
  ordering 72
  replacing 33
compile unit list 47
control record 198
control/RLD dictionary record format
  See control/RLD record write
CREATEW (create workmod function)
  parameter list 42
  return and reason codes 42
  syntax 41
CSECT
  identification record (IDR) format 200
CSECT (control section)
  aligning
    with ALIGNT function call 31
  changing 33
  data classes 2
  deleting 33
  expanding 33
  inserting 68
  ordering 72
  replacing
    with ALTERW function call 33

D

data area 289
data class, obtaining 109
data set
  specifying in STARTD function 92
data storing 73
DD statement
  alternative ddnames 170
DELETEW (delete workmod function)
  parameter list 43
  return and reason codes 43
  syntax 42
deleting external symbols
  with ALTERW function call 33
deleting sections 33
dialog token
  description 2
  obtaining 92
DLL support
  Exporting code 237
  Exporting data 237
  Importing code 237
    Non-XPLINK 238
    XPLINK 238
  Importing data 237
    Non-XPLINK 237
    XPLINK 237

DLLR
return and reason codes 45
syntax 44
DLLRename
parameter list 46
dynamic link library
DLL names inside IMPORT control statements 83

E

END object module record 190
ENDD (end dialog function)
parameter list 47
return and reason codes 47
syntax 46
entry name
changing 33
deleting 33
specifying 29
entry point
replacing 33
specifying AMODE
with ADDA function call 29
environmental options
CALLERID 173
LINECT 173
MSGLEVEL 173
PRINT 173
TERM 173
TRAP 173
WKSPACE 173
ESD
attribute assignment 216
behavioral attributes 212
continuation record 211
extended attributes 217
record 208
specifying elements 211
ESD (external symbol dictionary)
offsets 7
retrieving 52
ESD object module record 187
ESDID summary 211
example
attributes data record 356
IEWTPORT (transport utility) 349
item data record 357
transportable file format 351
transportable program body 354
transportable program structure 353
execution environment 20
user exit 177
exit
interface validation 180
message 178
save 178
setting return codes 179
user 177
EXITS 92, 180
exits user
specifying with STARTD function 92
external reference
changing 33
processing rules 38
replacing 33
external symbol
changing 33
deleting 33
replacing 33
external symbol definition
behavioral attributes 212
record 208

F

fast data access 109
code call interface 114
environment characteristics 110
error handling 127
parameters 110
using 109
formats
object module 191
FREEBUF
function 14
syntax 14
function calls 8

G

GETBUF function
syntax 15
GETC (get compile unit list function)
parameter list 49
return and reason codes 49
syntax 47
GETD (get data function)
parameter list 52
return and reason codes 51
syntax 50
GETE (get ESD data)
parameter list 56
return and reason codes 55
syntax 52
GETN (get names function)
parameter list 58
return and reason codes 58
syntax 56
GOFF
class names 205
conventions 205
generating DLLs 237
generating linkage descriptors 237
guidelines 203
incompatibilities 204
overview 203
record formats 204
record prefix 206
record types 203
restrictions 203

I

identification record data field 220
IDR
  data field 220
  format 1 221
IDR data
  extended format 221
  format 2 221
  format 3 221
IDRL 221
IDRU 221
IEWBFDA (obtain module data)
  return and reason codes 128
  service 109
  syntax 124
IEWBIND macro
  coding 23
  coding lists 93
  function summary 28
  parameter list 23
  return and reason codes 21
IEWBUFF macro
  coding 13
  function summary 13
IEWTFMT macro 351, 355
IEWTPORT (transport utility)
  allocating space 348
  convert a program library 349
  convert a program object 349
  convert a transportable program 349
  defining data 348
  errors 350
  examples 349
  JCL 347
  messages 349
  return codes 350
  selecting members 348
  SYSPRINT DD statement 348
  SYSUT1 DD statement 348
  SYSUT2 DD statement 348
  using 347
IMPORT (import variable function)
  parameter list 60
  return and reason codes 60
  syntax 59
import variable
  with IMPORT function call 59
INCLUDE (include module function)
  parameter list 67
  return and reason codes 65
  syntax 61
INITBUF function
  syntax 16
initial load segment 306
initializing buffers 16
input conventions 195
input record types
  formats 195
inserting sections 68
INSERTS (insert section function)
  parameter list 69
INSERTS (insert section function) *(continued)*
  return and reason codes 68
  syntax 68
interface validation exit 92, 180
  default exit routine 183
  post processing 183
INTFVAL 93, 180
invoking the API 1
  from high-level languages 20
  setting invocation environment 19
  using macros 20
  with IEWBIND macro 23
  with IEWBUFF macro 13
invoking the binder
  from a program 1, 169
  with the API 1
ITEM data type 356

J

JCL (job control language)
  coding
    IEWTPORT 347

K

keyboard
  navigation 361
  PF keys 361
  shortcut keys 361

L

LINK macro invoking binder 169
link pack area
  with RES option 38
linkage editor 195
LNAME option
  purpose 174
  specifying 88, 93
LOAD macro
  invoking
    binder 20
  invoking binder 169
  issued by IEWBIND macro 19
load module record formats 195
loading programs with binder API 69
loading the binder 19
LOADW (load workmod function)
  parameter list 71
  return and reason codes 70
  syntax 69

M

mainframe
  education xviii
MAPBUF function
  syntax 16
mapping buffers 16

message exit 92, 178
migration from OBJ to GOFF 203
MODLIB option
  purpose 175
  specifying 88, 93
module data, obtaining 109
multiple text classes 5
Multiple-text class modules 7
MVS linkage conventions 21

N

naming program modules 174
navigation
  keyboard 361
never-call option 86
non-reserved names 205
Notices 363

O

object module
  formats 191, 203
  including 61
  input conventions 185
  record formats 186
ordering sections 72
ORDERS (order section function)
  parameter list 73
  processing notes 72
  return and reason codes 73
  syntax 72
overlay program
  inserting sections 68
  INSERTS (insert section function) 68
  STARTS (start segment function) 98
overlay region
  assigning an origin 98
overlay segment
  assigning an origin 98

P

page alignment
  4KB boundary
    with ALIGNT function call 30
parameter list 23
  creating 20
  setting null values 23
passing lists to the binder API 93
PDS
  directory entry 289
PDS data area map
  cross reference 294
  on entry to STOW 289
  returned by BLDL 296
PDSE
  returned by DESERV 303
program module
  including 61
  saving 81
program module data
  location and order 7
program object
  accessing class information 306
  accessing with API 306
  accessing with external reference 306
programming interface information 365
pseudoregister
  changing 33
  deleting 33
  replacing 33
PUTD
  parameter list 77
  return and reason codes 76
  syntax 73

R

reason codes
  ADDA (add alias function) 30
  ALIGNT (align text function) 31
  ALTERW (alter workmod function) 35
  AUTOCall (API function) 37
  binder 99
  BINDW (bind workmod) 39
  CREATEW (create workmod function) 42
  DELETEW (delete workmod function) 43
  DLLR 45
  ENDD (end dialog function) 47
  GETC (get compile unit list function) 49
  GETD (get data function) 51
  GETE (get ESD function) 55
  GETN (get names function) 58
  IEWBFDA (obtain module data) 128
  IEWBIND macro 21
  IMPORT (import variable function) 60
  INCLUDE (include module function) 65
  INSERTS (insert section function) 68
  LOADW (load workmod function) 70
  ORDERS (order section function) 73
  PUTD (put data function) 76
  RENAME 79
  RESETW (reset workmod function) 80
  save user exit 179
  SAVEW (save workmod function) 83
  SETL (set library function) 88
  SETO (set option function) 90
  STARTS (start segment function) 99
record
  GOFF 204
  load module format 195
  RLD 222
  text continuation 220
record format load module IDR 200
record formats 203
record formats, object module 191
releasing buffers 14
relocation directory record 222
relocation record format 199
RENAME (rename symbolic references)
  parameter list 79

RENAME (rename symbolic references) *(continued)*
  return and reason codes 79
  syntax 78
rename DLL modules 44
replacing external symbols 33
replacing sections
  with ALTERW function call 33
reserved names 205
RESETW (reset workmod function)
  parameter list 81
  return and reason codes 80
  syntax 79
restricted no-call option 86
retrieving
  compile unit list 47
  ESD entries 52
  section names 56
  sections 50
  text 50
  workmod items 50
return codes
  IEWBFDA (obtain module data) 128
  IEWBIND macro 21
  IEWTPORT (transport utility) 350
RLD 222
RLD object module record 189
RLD record format 199

S

save exit 92
save exit routine 178
SAVEW (save workmod function)
  parameter list 86
  return and reason codes 83
  syntax 81
saving program modules 81
scatter table format 197
scatter/translation record format 197
screen readers
  accessibility 363
section
  definition 3
  inserting 68
  names 3, 23
  ordering 72
  retrieving 50
    names 56
section names 4
SETL (set library function)
  never-call option 86
  parameter list 88
  return and reason codes 88
  syntax 86
SETO (set option function)
  parameter list 91
  return and reason codes 90
  syntax 88
setting user exit return codes 178, 179
shortcut keys 361
side files
  saving 83
SMDE
  data area 303
  entry token section (normal) 305
  entry token section (system DCB) 305
  extensions 304
  file descriptor section 305
  format 303
  name section 304
  notelist section 305
  optional SMDE_SECTIONS 304
  primary name section 305
SNAME option
  purpose 175
  specifying 88, 93
specifying aliases and alternate entry points 28
specifying AMODE with ADDA 29
specifying binder options
  from a program 170
  with the API 173
    using SETO function 88
    using STARTD function 91
specifying call libraries 38
specifying substitute member names 78
STARTD
  ENVARS option 93
  PARMS option 93
STARTD (start dialog function)
  parameter list 97
  return and reason codes 96
  syntax 91
starting binder dialog 91
STARTS (start segment function)
  parameter list 99
  return and reason codes 99
  syntax 98
storing data with PUTD 73
STOW macro 289
string parameter
  example 21
structured-record data 218
supplying text information 218
SYM object module record 186
SYM record format 195
symbol (SYM) record format 195
symbol names
  purpose in API 5
  summary 5
SYSUTn 348

T

terminating binder dialog 46
text
  compression 222
  encoding types 222
  encryption 222
  object module record format 188
  record format 218
  retrieving 50

text continuation record 220
translation table format 197
transport utility 347
transportable file format
  alias data type 354
  attributes data type 355
  body 354
  example 350
  header 352
  item data type 356
  structure 350
  TP descriptor 353
  trailer 353
transportable program 347
  descriptor map 353
  structure 353

U

unitary (obtain module data)
  parameter list 127
unstructured-record data 218
user exit
  execution environment 177
  linkage conventions 177
  registers at entry 177
user exits 177
user interface
  ISPF 361
  TSO/E 361

V

varying character strings 23
version
  specifying in API 8
  specifying in binder 17
  summary 8

W

workmod
  binding 37
  creating 41
  deleting 42
  description 2
  item retrieving 50
  resetting 79
workmod element
  description 2
workmod token
  description 2
  obtaining 41
writing
  data, using PUTD 73
writing binder user exit routines 92, 177

X

XCTL macro 169

Z

z/OS Basic Skills information center xviii

Product Number: 5694-A01

Printed in USA

SA22-7644-14